CONTROL DE ACCESO INTELIGENTE A RECURSOS COMPARTIDOS DE GRANDES INSTITUCIONES

Óscar Caballero Rodríguez-Maribona
Irene Pastor Suárez
Fátima Rojo del Prado

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Grado en Ingeniería Informática

Curso Académico 2016/2017

Directores:

Luis Piñuel Moreno
José Ignacio Gómez Pérez

Autorización de difusión

Apellidos, Nombre

Madrid, a 16 de junio de 2017

Los abajo firmantes, matriculados en el Grado de Ingeniería Informática de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "*Control de acceso inteligente a recursos compartidos de grandes instituciones*", realizado durante el curso académico 16-17 bajo la dirección de Luis Piñuel Moreno y la co-dirección de José Ignacio Gómez Pérez en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Agradecimientos

Gracias a mis compañeros y tutores de este proyecto, por toda la ayuda que me habéis aportado durante todo el proyecto. También dar gracias a todos los amigos que, aun sin tener muchos conocimientos sobre el proyecto, se han procurado ayudarme con lo que han podido.

Irene

Gracias a mi compañero y amigo Pablo Aragón, por ayudarme a arrancar al principio de este proyecto. A mis tutores, por su paciencia y todo lo que nos han aportado, pero sobre todo a José Ignacio Gómez, por sus clases, sus tutorías y su disponibilidad y ayuda cuando también la he necesitado para planes futuros.

Fátima

Agradezco personalmente la ayuda de mis dos tutores, ya que sin ellos ni siquiera existiría este proyecto. Gracias también a toda esa gente que un día se preocupó de publicar un tutorial o guía en internet. Si no fuera por ellos seguiría atascado en el primer paso.

Óscar

Queremos dar las gracias, sobre todo, a nuestros profesores Luis Piñuel y José Ignacio Gómez del departamento de Arquitectura de Computadores y Automática, por darnos la oportunidad de formar parte de esta propuesta. Por su paciencia y su implicación y hacer posible que este proyecto saliera adelante.

Óscar, Irene y Fátima.

Índice general

Índice de figuras

Resumen

Este proyecto consiste en una red de Internet of Things (IoT) formada por pequeños dispositivos capaces de leer Tarejetas Universitarias Inteligentes (TUI). Estos dispositivos tienen la capacidad de recopilar información de estas tarjetas y enviarla a través de la red a una base de datos central. Los dispositivos consisten en Raspberry Pi equipadas con un lector de chips NFC que se encarga de leer las tarjetas. Esta información se envía a un gateway a través del protocolo LoRaWAN, que a su vez reenvía la información que recibe a un servidor usando MQTT. El servidor recoge los datos que recibe del gateway por MQTT y utiliza Kafka para procesarlos, y enviar la información pertinente a un Webservice con peticiones HTTP POST. Este Webservice permite comprobar que las peticiones son lanzadas correctamente proporcionando un servicio sencillo y dejando el proyecto preparado para posibles ampliaciones.
El objetivo final del proyecto es desplegar esta infraestructura en el campus de la universidad e instalar dispositivos en todas las aulas y algunas zonas con acceso restringido. Los dispositivos podrían leer la información del chip NFC presente en la tarjeta universitaria de la que disponen todos los alumnos y profesores para realizar tareas de control de asistencia a clase y de control de acceso a las zonas restringidas.

El desarrollo realizado finalmente se ha hecho a pequeña escala, es decir, con dos dispositivos, un gateway y una máquina virtual con recursos reducidos que actúa como servidor, pero está diseñado para poder crecer con facilidad y escalar al entorno ideado originalmente.

Palabras clave

Raspberry Pi, IoT, NFC, TUI, LoRa, MQTT, Kafka, WebService, Node-RED

Abstract

This project consists in an Internet of Things (IoT) network, composed by small devices that are capable of reading smart cards. These devices are able to collect some information from those cards and send it over the network to a central data base. The devices are single board computers (we used Raspberry Pi) equipped with NFC sensors that read the chip inside the smart cards. This information is sent to a gateway using the LoRaWAN protocol, who will then re-send the data it received to a central server over an MQTT connection. The server will process this data on a local-running Kafka client. The information that needs to be stored in the database will be forwarded to a web service running on the same machine, using HTTP POST requests. This Webservice allows the user to verify that the requests are being sent correctly, thus providing a simple service and giving the project room to grow.

The original concept of this project was to deploy this network infraestructure on the university campus, by installing the aforementioned devices in every classroom and some restricted access areas. The readers would be able to read the information contained inside the NFC tag that is part of the university's smart card that all students and professors have, thus being able to act as attendance and access control devices.

The final system has been developed at a small scale, that is, with two devices, a gateway and a low resource virtual machine that acts as a server. However, it has been designed to be able to grow easily and scale up to the environment that was originally devised.

Keywords

Raspberry Pi, IoT, NFC, TUI, LoRa, MQTT, Kafka, WebService, Node-RED

Capítulo 1

Introducción

La idea principal del proyecto es desarrollar un sistema basado en nodos de bajo coste, en nuestro caso hemos utilizado Raspberry, que sea capaz de leer el carnet de estudiante de los alumnos (Tarjetas TUI) a través de NFC, recibiendo una serie de datos de las tarjetas que se podrán utilizar posteriormente para diversas aplicaciones.

Una de las principales utilidades que podría tener el sistema es el control de asistencia a clase de los alumnos sin necesidad de que los profesores pierdan tiempo de sus clases en esta tarea. También podría ser utilizado para controlar el acceso a las instalaciones de la universidad, como garajes, instalaciones deportivas, laboratorios, etc.
La posibilidad de la implementación de nuestro proyecto en la universidad y las utilidades que podría aportar son las razones que nos han motivado a realizar este proyecto, ya que facilitaría varios aspectos de la vida diaria en la universidad para todos los que formamos parte de ella (alumnos, profesorado, personal administrativo, etc).

Como nuestro propósito es que este proyecto pueda ser implementado en la universidad, sería necesario instalar nodos en muchos puntos, y no todos los lugares donde querríamos implementar los nodos, tendrían acceso a WiFi, por lo que respecto al ámbito de la conectividad hemos elegido LoRa.

LoRa es una tecnología utilizada para comunicar dispositivos empleados en la denominada Internet de las Cosas (IoT). Es útil para redes de baja potencia y de área extensa. Los datos leídos se enviarán a través de un gateway LoRa. El gateway enviará los datos mediante MQTT, que es un protocolo de conectividad IoT. Es útil para conexiones en lugares remotos donde el ancho de banda es escaso, como podría ocurrir en su implementación en el campus universitario.

Los datos que se han recibido deben ser enviados a un servidor y hemos elegido Kafka, entre otras razones, por su alta escabilidad. Kafka es un sistema de almacenamiento productor/-subscriptor distribuido, particionado y replicado. Es un sistema muy eficiente respecto a la rapidez de las lecturas y escrituras, convirtiéndolo en una herramienta muy útil en nuestro proyecto y aportando un mayor número de posibles aplicaciones al proyecto.

Una vez Kafka recibe los datos, podemos notificárselo a los usuarios que lo deseen mediante una petición POST. Para ello, hemos desarrollado un Web Service sencillo que sea capaz de recibir y responder a las peticiones POST enviadas anteriormente. Este Web Service posibilita que se amplíe el número de aplicaciones que puede ofertar nuestro proyecto, ya que sería solo cuestión de desarrollar dentro del Web Service los servicios que queremos generar.

Algo que añadiría más funcionalidad al proyecto es que la comunicación sea bidireccional, es decir, que el nodo sea capaz de recibir la respuesta devuelta por el Web Service y en función de la respuesta, realizar otras acciones. Sería posible implementar esta opción, dado que el MQTT permite recibir las respuestas del Web Service, solo requeriría configurar MQTT de la forma correcta.

Todas estas tecnologías se explican, describiendo su funcionamiento, en los siguientes capítulos. La exposición sigue el mismo orden que el proceso de los datos, desde su recepción hasta su tratamiento y posible aplicación.

En la figura 1.1, podemos observar el flujo de dicho proceso, presentado en esta introducción.

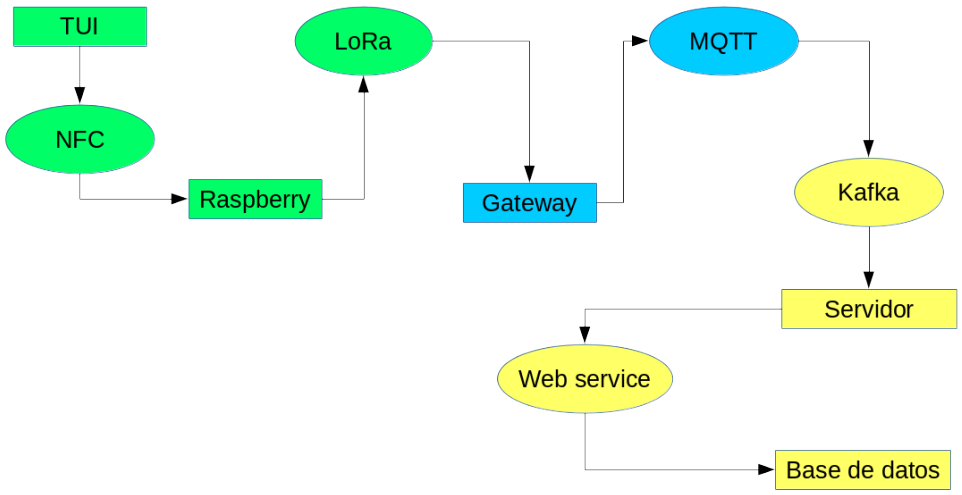

Figura 1.1: Flujo del proceso

Capítulo 2

Nodo

A la hora de empezar a desarrollar nuestro proyecto, lo primero de todo fue diseñar la parte hardware que compone el nodo lector. Nuestro prototipo consiste en un sistema basado en Raspberry Pi 3 capaz de leer tarjetas con tecnología NFC incorporada, concretamente tarjetas TUI. Una vez leídos los datos de las mismas, éstos son enviados a un servidor central, en nuestro caso, al CPD de la Universidad Complutense de Madrid en una versión final, donde se procesará dicha información para darle el uso oportuno.

Como se ha mencionado, se ha utilizado Raspberry Pi 3 para nuestro prototipo a la hora de diseñar el nodo, pero no es la única alternativa: la familia Arduino o la placa *BeagleBone* también son sistemas atractivos. En nuestro caso, se ha considerado Raspberry Pi 3 como un buen candidato puesto que se trata de un dispositivo con múltiples funcionalidades, fácil de usar y con un amplio soporte por parte de la comunidad. Asimismo, de cara a una futura selección de un nodo definitivo, teniendo en cuenta el coste económico, la compatibilidad del código desarrollado con la placa Raspberry Zero supone otro gran aliciente para su selección para realizar el prototipo. Dentro de este diseño aparecen dos tecnologías principales:

- Por un lado, NFC[1], una tecnología de comunicación inalámbrica de corto alcance que nos permitirá el intercambio de datos entre la tarjeta y el nodo lector. Para ello hemos usado EXPLORE-NFC, una placa de expansión NFC de alto rendimiento para la Raspberry.

- Y, por otro lado, LoRa. Puesto que la idea final es el manejo de muchos nodos en distintos puntos de la UCM, es posible que muchos de ellos no contaran con punto de red Ethernet o buena conexión WiFi. Es por ello que surge LoRaWAN[2] como solución de comunicación entre los distintos nodos y el servidor central, permitiendo conexiones bidireccionales seguras, de bajo consumo de energía y largo alcance de comunicación. En este caso, después de varias alternativas que explicaremos a lo largo del capítulo, finalmente se ha hecho uso del Microchip RN2483.

La figura 2.1 ilustra las diferentes etapas de procesamiento realizadas en el nodo, indicando la tecnología usada en cada una de ellas.

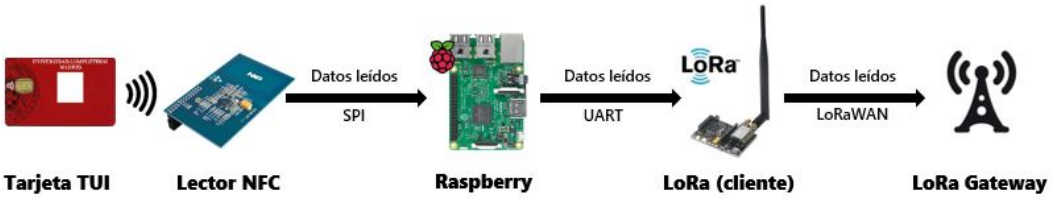

Figura 2.1: Infraestructura nodo lector

Con respecto al lenguaje de programación, tras varias alternativas y muchos quebraderos de cabeza, finalmente se ha optado por Python[3] a la hora de desarrollar el código que se encargará de devolver estos datos. Python es un lenguaje de programación interpretado que soporta distintos modelos de programación.

La razón principal por la que se ha elegido Python es, básicamente, lo mucho que facilita el desarrollo. Muchos trabajos que requieren cientos de líneas de código, con Python se realizan en pocas sin perder nada de claridad. Otra de las facilidades que nos aporta este lenguaje de programación es su librería estándar, que soporta gran diversidad de tareas.

2.1. Raspberry Pi

Como se ha mencionado anteriormente, nuestro nodo está compuesto -entre otros dispositivos- por una Raspberry Pi[4], en concreto el modelo Raspberry Pi 3 Model B. Es un computador

de placa única, un pequeño ordenador de bajo coste que permite realizar varias tareas, lo suficientemente potente como para funcionar como un ordenador personal. Su software es *open source* y cuenta con Raspbian[5] como sistema operativo, una versión adaptada de Debian (cuyo núcleo es Linux) aunque es capaz de soportar muchos otros.

Figura 2.2: Raspberry Pi 3 Model B

2.1.1. Características de Raspberry Pi

GPIO (General Purpose Input/Output) Una de las características más importantes de Raspberry son los pines GPIO[6] con los que cuenta. Se trata de pines entrada/salida y son la interfaz física que conecta la Raspberry con múltiples dispositivos, permitiendo añadir gran cantidad de funcionalidad al sistema que queramos diseñar. De los 26 pines con los que cuenta la placa, 17 son de GPIO mientras que el resto se corresponden con pines de alimentación y tierra.

Gracias a estos pines se pueden conseguir diferentes tipos de aplicaciones con nuestra Raspberry Pi, desde el diseño de una consola utilizando pantallas LCD, hasta controles de refrigeración con sensores de temperatura. En este proyecto, GPIO nos ha permitido conectar nuestra Raspberry al lector NFC que recogerá los datos de las tarjetas, así

como a nuestro dispositivo LoRa que permitirá enviar los datos recogidos al resto de nodos a través de su modelo de comunicación.

SPI (Serial Peripheral Interface) Raspberry Pi cuenta con SPI[7], un bus de comunicación en serie que trabaja de forma síncrona y cuya función principal es el establecimiento de comunicación de corta distancia. Se trata de un modelo de comunicación maestro-esclavo donde sólo puede existir un único maestro. La transmisión y sincronización de datos se realiza por medio de 4 señales:

- SCLK: Serial Clock. El que marca el envío de datos, a cada señal de reloj se envía un bit. Ubicada en el pin físico 23.
- MOSI: Master Output Slave Input. El envío de datos se realiza de maestro a esclavo. Ubicada en el pin físico 19.
- MISO: Master Input Slave Output. El envío de datos se realiza de esclavo a maestro. Ubicada en el pin físico 21.
- SS: Slave Select. Señal utilizada para seleccionar un esclavo. Ubicada en los pines físicos 24 y 26.

En nuestro prototipo, EXPLORE-NFC es el dispositivo que hace uso de este bus, por lo que es muy importante activar el protocolo SPI en la Raspberry para que nuestro lector pueda recoger y datos y volcarlos en ella.

UART (Universal Asynchronous Receiver-Transmitter) UART[8] también se encuentra integrado en la placa Raspberry. Se trata de un puerto de comunicación asíncrona (a diferencia de SPI) que permite la transmisión y recepción de datos en serie entre dispositivos. El transmisor envía los bytes de datos bit a bit, mientras que el receptor se encarga de reensamblar los bits que va recibiendo hasta formar los bytes completos que se enviaron. En concreto, los pines de transmisión y recepción UART de Raspberry están ubicados en los pines (físicos) GPIO 8 (TXD) y 10 (RXD) respectivamente y

trabajan a 3.3V.

Por tanto, el módulo LoRa hace uso del puerto UART para recibir los datos y enviarlos al gateway, mientras que EXPLORE-NFC utiliza el bus SPI.

2.1.2. Ventajas y utilidades de Raspberry Pi

Entre otras alternativas que pueden resultar más potentes, Raspberry Pi presenta muchísimas ventajas y facilidades en cuanto a desarrollo. Seguidamente detallamos algunas de ellas por las cuales se ha elegido Raspberry como base de nuestro sistema.

Coste

A pesar de tratarse de un simple prototipo, la finalidad de este proyecto, a largo plazo, es la implantación de este tipo de sistemas en distintos puntos de la Universidad Complutense de Madrid. Si, por ejemplo, sólo tenemos en cuenta las 26 facultades con las que cuenta actualmente la universidad y contamos con una media de 30 aulas por facultad, como mínimo se necesitarían alrededor de unos 780 nodos, sin contar con otras muchas instalaciones. Esto significa que el coste a la hora de implantar este sistema podría incrementarse y una de las muchas ventajas con las que cuenta la Raspberry es su bajo coste en el mercado, aun ofreciendo una gran cantidad de recursos, pudiendo reducir así la inversión realizada.

Otra alternativa de bajo coste podría ser Arduino; sin embargo éste no puede ejecutar todo un sistema operativo, funciona a una velocidad de reloj mucho más lenta que Raspberry y cuenta con una memoria RAM insignificante, lo que apenas nos permitiría realizar varias tareas a la vez. Además, el coste de una placa Arduino no es muy inferior al de una Raspberry Pi 3. Se puede obtener más información sobre Arduino en su página web.

Como se mencionó en la introducción, otra ventaja de seleccionar Raspberry Pi 3 como prototipo es la existencia de Raspberry Pi Zero y Raspberry Pi Zero W: por un precio de 5€ (11€ para la versión *W* que incluye Wifi y Bluetooth) tenemos una placa compatible tanto a nivel de pines como de sistema con nuestro prototipo, con lo que la transición sería prácticamente automática.

Alcance

Cuando hablamos de las facilidades que nos aporta Raspberry en cuanto a soporte y alcance, nos referimos a los pines GPIO incorporados en ella, cuyo comportamiento es controlable para los desarrolladores. En nuestro caso, GPIO nos ha permitido conectar tanto el módulo NFC para la lectura de tarjetas, como el módulo LoRa, creando nuestro propia infraestructura para el envío de datos al servidor. Pero ésto es sólo una aplicación de muchas, lo 40 pines de entrada/salida permiten comunicar la Raspberry con el mundo exterior, abriendo puertas al amplio sector IoT (más información en Internet of Things), por ejemplo, la entrada puede venir de un sensor, o incluso de una señal de otro dispositivo. La salida también puede ser desde el encendido y el apagado de LEDs, hasta el envío de señales o datos a otro dispositivo. Con Raspberry Pi se puede conseguir desde una consola con un joystick y botones, hasta una pequeña estación meteorológica recogiendo datos a través de sensores.

Conectividad

Los anteriores modelos de Raspberry Pi sólo podían conectarse a través de un cable Ethernet o bien mediante adaptadores inalámbricos USB. Sin embargo, el modelo Raspberry Pi 3 Model B de nuestro prototipo incluye una tarjeta de red WiFi inalámbrica que facilita mucho las cosas. No sólo a la hora de desarrollar, como cuando queremos trabajar desde nuestra propia computadora conectándonos al dispositivo, sino que de cara al proyecto, también se plantea la posibilidad de utilizar WiFi como modelo de comunicación entre los nodos y el servidor, ampliando las alternativas de intercambio de datos y facilitando el desarrollo de proyectos IoT.

2.1.3. Primeros pasos

Uno de los primeros pasos a la hora de empezar a desarrollar nuestro proyecto fue preparar nuestra Raspberry para su uso. Lo primero de todo fue instalar el sistema operativo, para ello es necesario descargar la imagen del mismo. En nuestro caso utilizamos Raspian

Jessie, descargamos la imagen en formato .zip a través de la página oficial de Raspberry[9] y descomprimimos el archivo en nuestra tarjeta microSD.

Al tratarse de un archivo .img, es necesario escribir la imagen en la tarjeta microSD (previamente formateada), por lo que utilizamos *Win32DiskImager*, una aplicación de escritorio que nos permite realizar dicha operación.

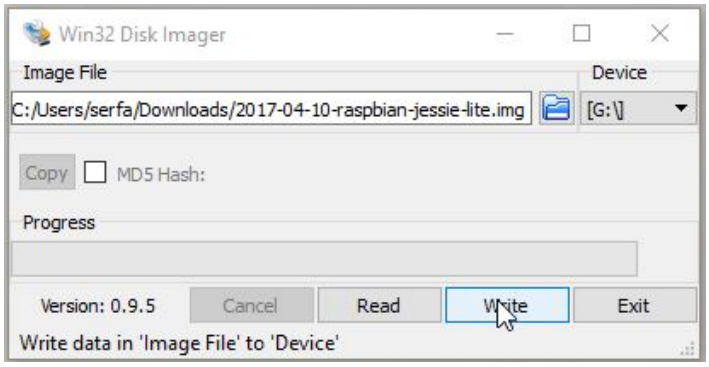

Figura 2.3: Win32DiskImager. Write option

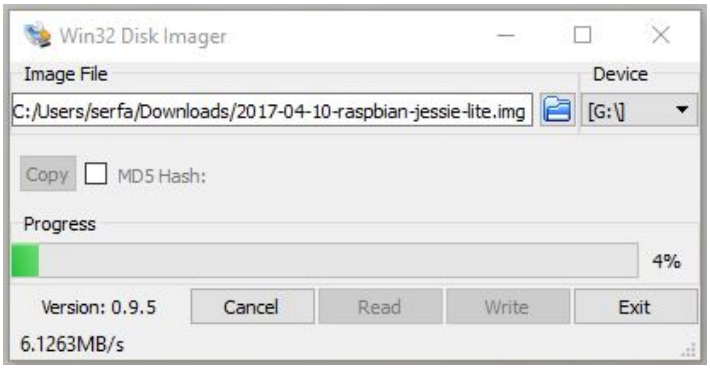

Figura 2.4: Win32DiskImager. Writing image

Tras tener preparada la tarjeta microSD con Raspbian, introducimos la tarjeta en nuestra Raspberry Pi, con todos los componentes necesarios conectados y, por último, conectamos a la fuente de alimentación. Automáticamente, las Raspberry Pi se enciende y comienza a configurar el sistema operativo. Una vez terminado el proceso de arranque, iniciamos sesión a través de la línea de comandos, cuyas credenciales iniciales son: `Usuario: pi` `Contraseña: raspberry`

Opcionalmente, si queremos iniciar la interfaz gráfica, introducimos el comando `startx`. Nos conectamos a nuestra red WiFi y apuntamos la dirección IP asignada a la Raspberry con el comando `ifconfig` para los siguientes pasos.

Uno de los primeros cambios de configuración de nuestra Raspberry Pi, es activar el protocolo SSH (más información en Protocolo SSH, para poder trabajar con ella desde nuestra computadora y ahorrarnos la necesidad de conectar todos estos dispositivos. Par ello, desde la línea de comandos realizamos los siguientes pasos:

1. Introducimos `sudo raspi-config` desde la terminal
2. Seleccionamos `Advanced Options`
3. Seleccionamos `SSH`
4. Elegimos la opción `Yes`
5. Seleccionamos `Ok`
6. `Finish`

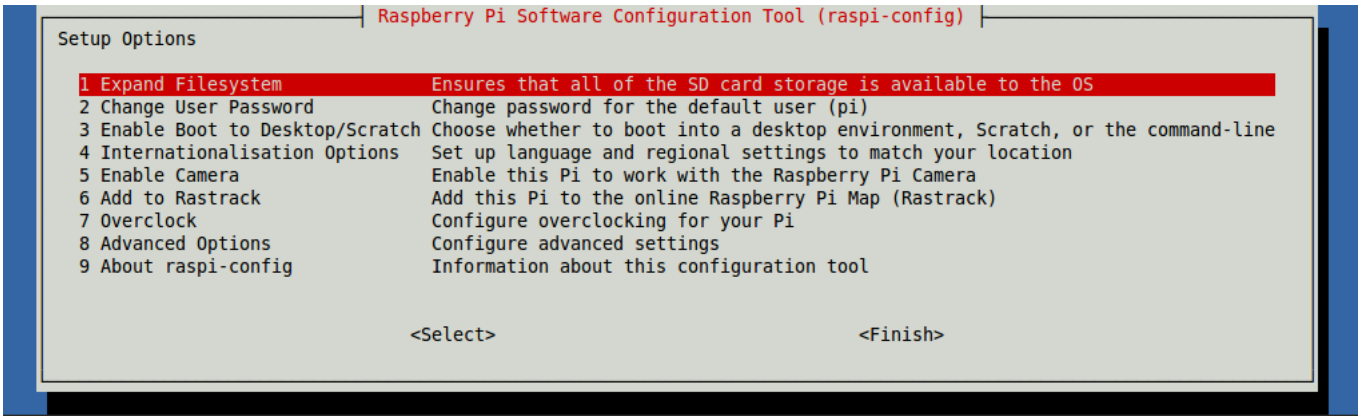

Figura 2.5: Raspberry Pi Software Configuration Tool (raspi-config)

Por último, necesitamos un cliente SSH con el que conectarnos desde nuestra máquina a la Raspberry. Existen muchas alternativas, nosotros hemos utilizado PuTTy, uno de los clientes SSH más comunes para Windows y que podemos descargar a través de su página

web. Una vez en la aplicación, introducimos la IP de nuestra Raspberry Pi -que apuntamos anteriormente- y pulsamos el botón `Open`.

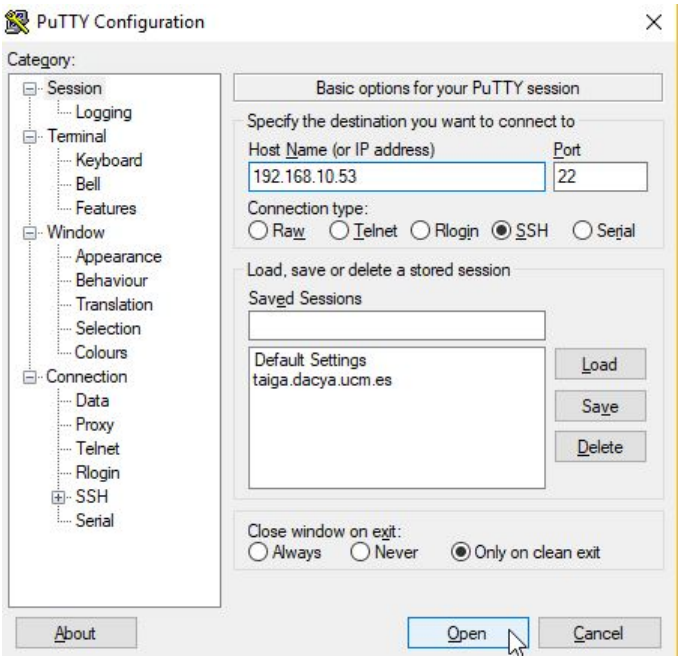

Figura 2.6: PuTTy. Aplicación de escritorio

Si la conexión es satisfactoria, aparecerá una ventana de comandos que nos pedirá las credenciales del dispositivo al que nos queremos conectar para iniciar sesión. Iniciamos sesión en nuestra Raspberry y ya podemos comenzar a trabajar con ella desde nuestro ordenador.

2.2. NFC

Después de dejar preparada nuestra Raspberry, el siguiente paso a la hora de construir el nodo lector fue incluir el dispositivo NFC a nuestro sistema. Este dispositivo es el encargado de recoger datos de las Tarjetas TUI gracias a su tecnología NFC, datos que posteriormente serán enviados al servidor central, como comentábamos al principio del capítulo.

NFC son las siglas correspondientes a *Near Field Communication*[1] una tecnología de comunicación inalámbrica, de corto alcance y alta frecuencia que permite el intercambio de datos entre dispositivos. Se trata de una comunicación basada en radiofrecuencia.

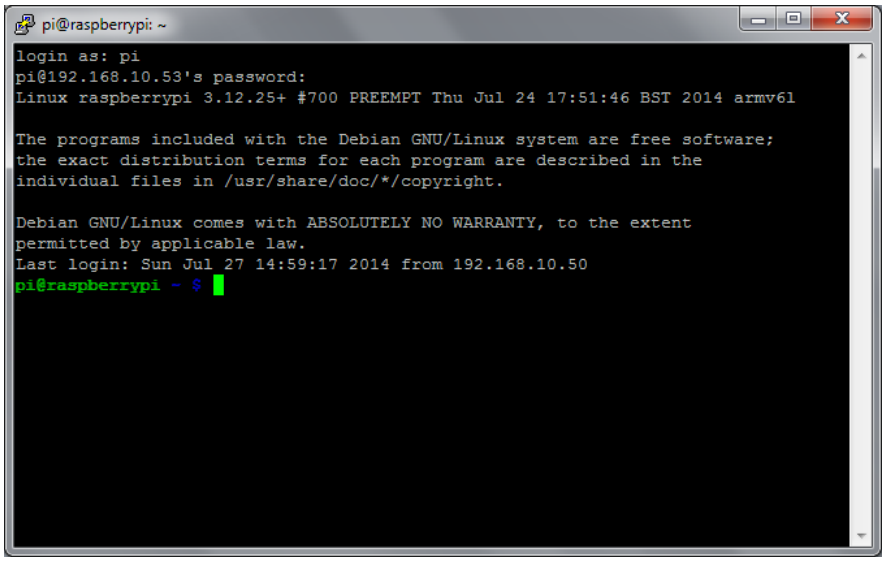

Figura 2.7: PuTTy. Consola Raspberry

En términos generales, lo que NFC proporciona a sus usuarios finales son:

- Conexiones sencillas: con el simple gesto de acercar los dispositivos NFC, se establece la conexión entre ellos.
- Transacciones rápidas: está diseñado para intercambio de datos de pequeño tamaño por lo que una vez se establece la conexión, las transacciones resultan instantáneas al usuario.

NFC soporta tres modos de operación:

- **Lectura/Escritura**: permite a los dispositivos NFC leer información almacenada en etiquetas NFC de bajo costo incrustadas en carteles y pantallas inteligentes. Esto puede resultar una gran herramienta de marketing para las empresas.
- **Peer-to-peer (P2P)**: permite que dos dispositivos NFC se comuniquen entre sí para intercambiar información y compartir archivos, con el simple gesto de acercarlos.
- **Card Emulation**: permite que los dispositivos NFC actúen como tarjetas inteligentes, comunicándose con un lector externo, permitiendo a los usuarios realizar transacciones como compras, ticketing y control de acceso a instalaciones.

Como explicábamos, nuestro nodo lector se basa en una Raspberry Pi 3 Model B y, entre otras cosas, se encarga de leer datos de tarjetas universitarias. Esta función, por tanto, es llevada a cabo por un lector NFC conectado a nuestra Raspberry mediante los pines GPIO. En concreto, hemos utilizado el modelo PNEV512R[10] de EXPLORE-NFC proporcionado por NXP. Se trata de una tarjeta de expansión NFC de alto rendimiento para la Raspberry Pi y compatible con los tres modos de comunicación: Lectora/Escritura, P2P y Card Emulation.

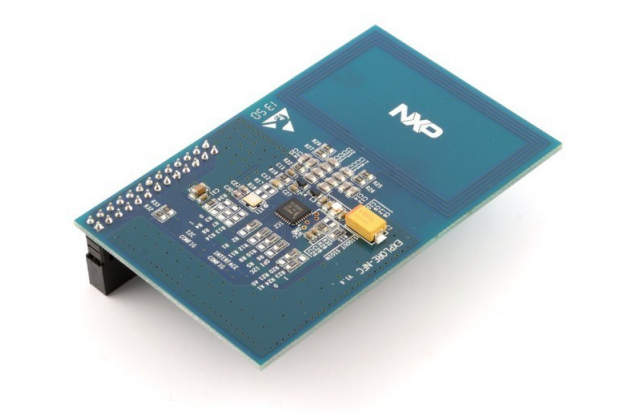

Figura 2.8: EXPLORE-NFC PNEV512R

2.2.1. Ventajas y utilidades de la tecnología NFC

Velocidad

La conexión entre dispositivos no necesita emparejamiento previo y la función principal de NFC consiste en el intercambio de pequeña información. Por tanto, a pesar de que la tasa de transferencia NFC ronda los 424 kbit/s, ésta resulta instantánea para el usuario cuando los dispositivos se acercan. A diferencia, Wi-Fi o Bluetooth te permiten estar incluso en habitaciones distintas y transmitir grandes cantidades de datos. Sin embargo, si queremos conectar por primera vez nuestro móvil al manos libres del coche, para conseguirlo sin NFC tendríamos que activar el Bluetooth del teléfono, rastrear dispositivos, conectar con ellos, introducir la clave, etc. Con NFC sería tan sencillo como deslizar el móvil sobre el manos

libres, omitiendo el resto de pasos.

Como se ha mencionado, NFC no está pensada para transferir ficheros de gran tamaño sino para identificar o validar a otro dispositivo y ésta es la ventaja y utilidad que tiene dentro del contexto de nuestro proyecto. Buscamos mandar una información mínima, en tiempo real al servidor de la UCM. Si usáramos nuestro nodo como un lector instalado en las aulas de la facultad, donde cada alumno tuviera que registrar su entrada a las clases, ¿cuánta cantidad de alumnos estarían involucrados? Gracias a NFC no habría esperas, sólo haría falta acercar la tarjeta al lector para identificarse, ahorrando gran cantidad de tiempo y esfuerzo a la hora de sincronizar los dispositivos involucrados.

Seguridad

El alcance NFC es muy reducido, pues se mueve como máximo en un rango de 20 cm. Pero, aunque esto parezca un punto débil, esto también ayuda a aumentar la seguridad de intercambio de información, evitando que el dispositivo se conecte a otros no deseados. Además, NFC ofrece funciones integradas para soportar aplicaciones seguras.

Intuitivo y versátil

Cuando hablamos de intuitivo, nos referimos a que el modo de uso de los dispositivos NFC no puede ser más sencillo: un simple toque a corta distancia y la interacción se realiza.

Además, a pesar de que la introducción de la tecnología durante los últimos años ha sido un tanto lenta, hoy en día las distintas aplicaciones que se pueden conseguir con NFC son cada vez mayores. Es por eso que NFC es ideal para cualquier tipo de industria, sector y uso. Gracias a esta tecnología podemos encontrar diferentes tipos de soluciones para intercambio de datos, control de acceso, transporte o incluso funcionar como método de pago. Estas son algunas de las más conocidas:

- **Intercambio de datos entre smartphones**: ahora es posible intercambiar datos mediante P2P entre dos dispositivos NFC, como intercambio de contactos, o de cualquier tipo de archivo.
- **Acceso físico**: con el simple gesto de acercar el smartphone la puerta del coche,

se abre y es posible activar funcionalidad previamente programada, como el GPS o escuchar música acercando el teléfono a la radio.

- **Identificación**: en grandes compañías, no sólo controlan los registros de sus empleados a través de sus tarjetas, sino que éstos a su vez tienen acceso o no a distintas instalaciones gracias a ellas, o incluso al uso de impresoras de la compañía. Algunas compañías aéreas plantean la posibilidad de hacer check-in con nuestros teléfonos gracias a esta conectividad, algo que sin duda facilitaría y agilizaría el proceso.

- **Pagos**: algunos dicen que el móvil sustituirá al monedero. Cada vez hay más usuarios que utilizan esta funcionalidad, acercando el smartphone a un dispositivo NFC ya se puede pagar la compra en un supermercado.

2.2.2. Tarjetas TUI

El propósito de este proyecto es leer los datos del carnet universitario del que disponen los alumnos de la Complutense. Actualmente este carnet se trata de una Tarjeta Universitaria Inteligente (TUI)[11], una credencial que almacena información personal en un chip e incluyen otro de proximidad. A través de éste último podemos leer información de la misma y hacer uso de otros servicios como:

- Firma electrónica: identificación y cifrado en Internet.

- Control de acceso: puede utilizarse para acceso a instalaciones universitarias.

- Prestación de material: permite gestionar préstamos de libros de la biblioteca o de material de laboratorio.

- Monedero o tarjeta débito: puede utilizarse como monedero con o sin vinculación a cuentas bancarias. También ofrece descuentos en comercios asociados con la universidad.

Estas tarjetas son de tipo MIFARE Classic 4k[12] una expansión de las MIFARE Classic 1k[13] que incrementa el espacio de memoria. La siguiente tabla muestra las características y organización de memoria de las dos tarjetas:

	MIFARE Classic 4K	**MIFARE Classic 1k**
UID	7 bytes (Bloque 0, Sector 0)	
Memoria	4KB	1K
Bloques	256 bloques: 32 sectores de 4 bloques 8 sectores de 16 bloques	64 bloques: 16 sectores de 4 bloques
Claves	A y B almacenadas en el último bloque de cada sector	

Cuadro 2.1: Características MIFARE Classic.

Como podemos ver en la tabla, la organización de la memoria difiere entre los dos tipos y esta diferencia ha supuesto algunos problemas a la hora de acceder a la información de las tarjetas TUI.

Cuando el lector NFC reconoce una tarjeta, automáticamente se devuelve el UID de la misma. Sin embargo, para poder acceder al resto información almacenada en los bloques, es necesario realizar una autenticación, ya sea para leer o escribir. Esta autenticación consiste en leer una de las claves A ó B para poder acceder al contenido de la tarjeta.

La biblioteca utilizada de Python (`nxppy`), actualmente sólo proporciona funcionalidad para las tarjetas de tipo MIFARE Classic 1k, por lo que incluye soporte para realizar la autenticación de las mismas y acceder a sus bloques. Dado que la organización de la memoria varía en las MIFARE Classic 4k, el soporte de autenticación que proporciona la librería no es compatible con ellas, por lo que no es posible autenticarlas y acceder al resto de contenido. A pesar de realizar varios intentos de añadir nuestra propia funcionalidad a la librería, no se ha conseguido autenticar dichas tarjetas.

Figura 2.9: Tarjeta TUI UCM

2.2.3. Instalación y configuración

A la hora de incorporar el dispositivo NFC a nuestra Raspberry, con ésta apagada, conectamos la placa PNEV512R a nuestra Raspberry mediante los pines GPIO, teniendo cuidado de colocarla en el modo correcto. En la siguientes figura se puede ver como quedan conectados:

Figura 2.10: Raspberry Pi 3 Model B y PNEV512R

Como ya teníamos preparada y configurada nuestra Raspberry, sólo hacía falta conectarla a la fuente de alimentación e iniciar sesión en nuestro sistema. Nuestro lector NFC se comunica con la Raspberry a través del bus SPI por lo que una vez en el sistema, es necesario activar el protocolo en la Raspberry. Para ello, al igual que hicimos con el protocolo SSH, desde la línea de comandos seguimos los siguientes pasos:

1. Introducimos `sudo raspi-config` desde la terminal

2. Seleccionamos `Advanced Options`

3. Seleccionamos `SPI`

4. Elegimos la opción `Yes`

5. Seleccionamos `Ok`

6. `Finish`

Por último, para aplicar los cambios, reiniciamos el sistema ejecutando el comando `sudo reboot`.

Una vez integrado nuestro lector NFC con la Raspberry Pi, instalamos el software necesario para empezar a realizar pruebas y continuar con el desarrollo de nuestro proyecto. Para ello, nos descargamos el software disponible en la web de NXP[14] del dispositivo, en concreto la versión de Neard EXPLORE-NFC para Raspbian Jessie.

Para esta parte se necesita una memoria USB o pendrive en el que descargar el fichero que contiene el software. Posteriormente lo descomprimimos en la Raspberry e instalamos los paquetes Debian incluídos con `dpkg` ejecutando el siguiente comando:

```
sudo dpkg i libneardal0VERSIONarmhf.deb neardexplorenfcVERSIONarmhf.deb
```

Tras realizar los pasos anteriores, comprobamos que nuestro lector NFC funcionaba correctamente. Para comprobarlo, probamos algunos de los ejemplos que nos facilitan. En el caso de la lectura de tarjetas ejecutamos el comando `explorenfc-basic` el cual mostraba información de la tarjeta por pantalla. Por otro lado, el de escritura `explorenfc-basic -w 'Hola Mundo'` escribía el mensaje en un bloque de la tarjeta y cuando ésta era leída de nuevo, nos mostraba también la información con el mensaje.

2.2.4. Datos Recogidos

Una vez tuvimos nuestro lector NFC funcionando, empezamos a plantear qué datos se recogerían de las tarjetas TUI. Después de varias alternativas, finalmente decidimos que para nuestro prototipo leeríamos:

- UID: Identificador de la tarjeta. En este caso, las tarjetas TUI son tarjetas de tipo MIFARE Classic 4K, cuyo identificador único viene representado en 7 bytes.
- Contador: un contador añadido a cada tarjeta de tal manera que, en caso de duplicación de las mismas, nos permite identificar la original.

Por otro lado, se recoge la fecha y hora actual en una variable tipo time que nos devuelve el momento de lectura del dispositivo. El formato en que se muestra es `YY-MM-DD HH-MM-SS`.

Antes de comenzar a escribir nuestro programa, fue necesario añadir la librería `nxppy`, una envoltura de Python para la interconexión con la placa EXPLORE-NFC. Contiene la librería NFC Reader[16] que ofrece NXP, escrita en C y que proporciona una capa para detectar etiquetas MIFARE, leer su UID y realizar operaciones de lectura/escritura. El proyecto nxppy se encuentra actualmente en Github[15]. Es importante destacar que para la utilización de esta librería sólo las versiones 2.7 y 3.4 de Python son compatibles y que el protocolo SPI esté activado, como hemos explicado en secciones anteriores.

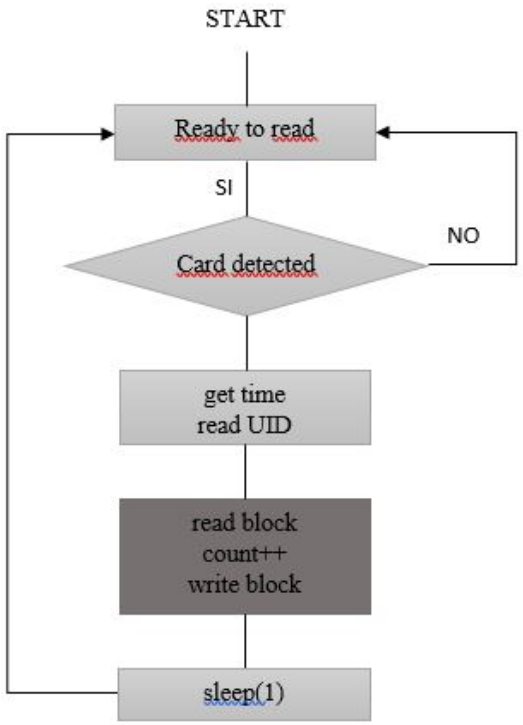

Figura 2.11: Flujo ejecución NFC.

La instalación de la librería se realiza con el comando `pip` y puede llevar a cabo unos minutos: `pip install nxppy`

A partir de aquí, toda la parte del nodo lector quedaba montada y configurada. Solo faltaba fijar el código que se encargaría de leer el información de la tarjeta. Como bien se ha comentado, Python ha facilitado muchísimo la implementación del código que se basa en lo siguiente:

Por un lado, importa las librerías `nxppy` (que aporta la funcionalidad para NFC) `time` (para conseguir la fecha y hora del momento de lectura)

La función del resto del código es esperar a que detecte una dispositivo NFC, en nuestro caso tarjetas TUI. Una vez detecta el dispositivo, accede a su UID y seguidamente guarda la fecha y hora en que leyó el dispositivo. Y así hasta que la ejecución del código sea interrumpida. En el diagrama de la figura 2.11 podemos ver el flujo del código, la parte sombreada sólo tiene lugar si la tarjeta leída es de tipo MIFARE Classic 1k, donde se lee y escribe el contador del que hablábamos anteriormente. En la figura 2.12 se muestra un ejemplo de salida por pantalla tras la ejecución del código.

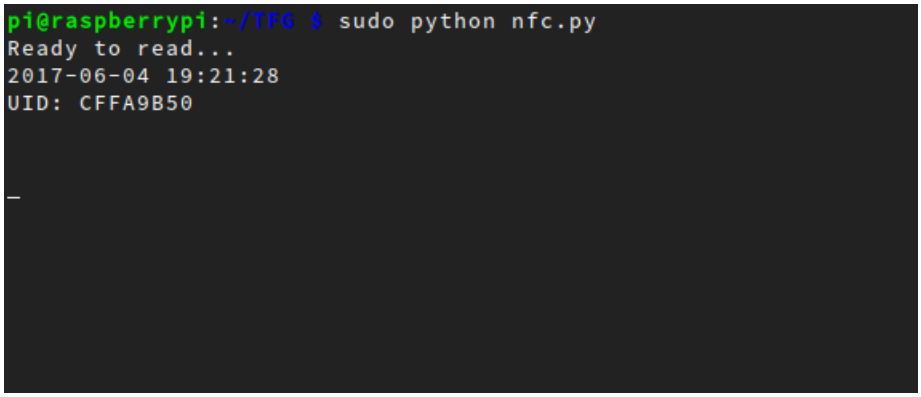

Figura 2.12: Ejemplo lectura NFC.

2.3. Cliente LoRaWAN

2.3.1. Descripción del protocolo

La segunda fase de la comunicación consiste en el envío de los datos recogidos por el lector al gateway, que actúa como punto central de la comunicación enñtre todas las Raspberry Pi y el servidor. Para esta fase de la comunicación se ha decidido, por una serie de razones detalladas más adelante, utilizar el protocolo LoRaWAN.

LoRa es una técnica de modulación desarrollada en 2008, y adquirida más tarde por Semtech, que fue diseñada con el objetivo de ofrecer una comunicación de largo alcance y

bajo consumo. Estas dos características convierten a LoRa en una tecnología ideal para una red de IoT como la de este proyecto, aunque en este caso el consumo de energía no es un problema tan crítico ya que los nodos están conectados a la corriente constantemente, y no dependen de una batería.

LoRa en sí es una tecnología de muy bajo nivel, de modo que para construir una infraestructura de red se utiliza LoRaWAN. LoRaWAN es un protocolo LPWAN (Low Power Wide Area Network) basado en LoRa que ofrece servicios de MAC en la capa de enlace de datos (OSI nivel 2), lo que lo convierte en una solución lo suficientemente básica como para dar flexibilidad de implementación al usuario, pero encargándose de tareas fundamentales como la estabilidad o la seguridad.

Una red LoRaWAN básica está estructurada en forma de estrella, en la que una serie de nodos se conectan a un gateway central. En la mayoría de las configuraciones, aunque no necesariamente, el gateway está conectado a la red y envía los datos que recibe de los nodos de LoRaWAN a un servidor en paquetes de cualquier otro protocolo (TCP, UDP, MQTT etc). Este servidor suele ser el encargado de almacenar o procesar los datos de la manera apropiada y puede, opcionalmente enviar una respuesta al gateway para que la retransmita, de nuevo a través de LoRaWAN, a los nodos.

En una configuración más compleja en la que participa un número elevado de nodos o que se extiende en un área más amplia, es necesario introducir más de un gateway en la red.

En el primer caso, la adición de más gateways permite repartir la carga de procesamiento, ya que independientemente del modelo que se utilice, siempre existe un límite de conexiones que un gateway puede mantener activas antes de que el procesador o el hardware de red se vean sobrecargados. Si ocurre esto y se ve necesario introducir más de un gateway, habrá un problema de interferencias entre ellos, ya que más de uno podrá recibir la información enviada por cada nodo. En este caso será necesario configurar cada gateway para aceptar las transmisiones de algunos nodos e ignorar las demás, dado que no hacerlo no solo no solucionaría el problema de sobrecarga que se intentaba evitar sino que además produciría

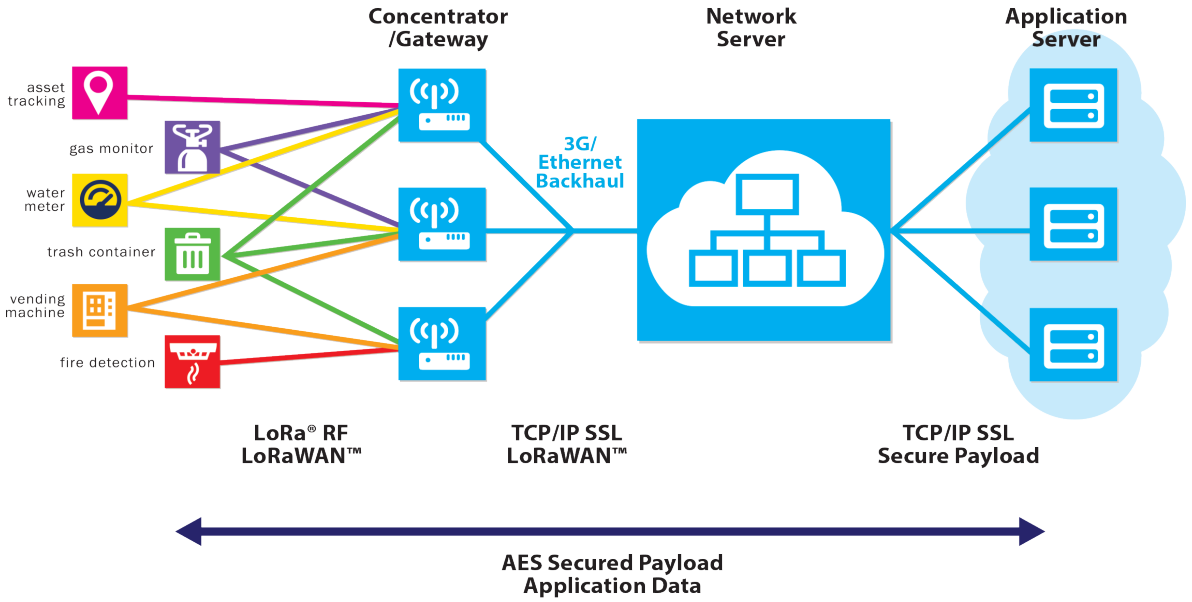

Figura 2.13: Arquitectura de LoRaWAN.
Fuente: Semtech Corporation

datos duplicados en el servidor, puesto que varios gateways enviarían la misma información repetida.

En el segundo caso, idealmente no se produciría este problema porque los gateways se encontrarían en zonas distintas y no interferirían entre ellos.

Es importante notar, que el protocolo LoRaWAN está diseñado para IoT, y por lo tanto asume que la información que se transmitirá a través de la red será de un tamaño relativamente pequeño. Por esta razón y con la intención de reducir el consumo de energía de los nodos, el ancho de banda típico de una red LoRaWAN no excede de los 50 Kb/s. Esto hace que este protocolo no sea la mejor elección para escenarios en los que sea necesario transmitir una gran cantidad de datos a través de la red.

La especificación oficial de LoRaWAN establece el ancho de banda en un rango de entre 0,3 y 50 Kb/s, aunque la velocidad real varía de nodo a nodo y se establece dinámicamente en el lado del servidor usando una técnica de Adaptive Data Rate (ADR).

Tipos de nodos

En función de la aplicación que se le quiera dar a la red LoRaWAN, se pueden establecer tres tipos de optimización para cada nodo:

Clase A Dispositivos optimizados para bajo consumo. En estos casos son los nodos los que deben iniciar la comunicación, enviando paquetes de datos al servidor en un intervalo de tiempo predeterminado y escuchando los paquetes de respuesta durante otro intervalo también predeterminado. Este modelo de comunicación implica que los dispositivos solo tienen que estar activos durante cortos intervalos de tiempo para así ahorrar más energía, pero también significa que las transmisiones se realizan con más retraso y más lentamente, ya que si la información necesaria no ha podido ser enviada por completo en uno solo de los intervalos de tiempo preestablecidos, ambos dispositivos deberán esperar hasta el siguiente para poder continuar con la comunicación.

Clase B Los dispositivos de clase B reducen los problemas de latencia de los de clase A permitiendo que el gateway también pueda iniciar la comunicación. De esta manera, si al gateway aún le quedan datos por enviar, puede mandar una señal de activación (beacon) al nodo para que se ponga en modo de escucha de paquetes y pueda recibir el resto de la información. Esto reduce la latencia de la comunicación respecto al modelo A, pero no la elimina por completo, ya que ésta sigue realizándose en intervalos de tiempo y no continuamente. Además incrementa el consumo de energía de los dispositivos, que deben estar activos durante más tiempo, por lo que puede no ser una solución ideal para nodos que dependan de una batería.

Clase C Los dispositivos de clase C reducen al máximo posible la latencia de la comunicación, haciendo que los nodos estén siempre esperando recibir paquetes del gateway, y solo dejan de escuchar cuando cambian a modo transmitir. Esto permite que el gateway pueda enviar datos en casi cualquier momento, reduciendo enormemente la latencia respecto a las otras dos clases de dispositivos. Los dispositivos de clase C

están, idealmente, conectados a una fuente de alimentación constante, ya que el consumo de energía que supone estar siempre activos es mucho mayor que en los otros dos casos.

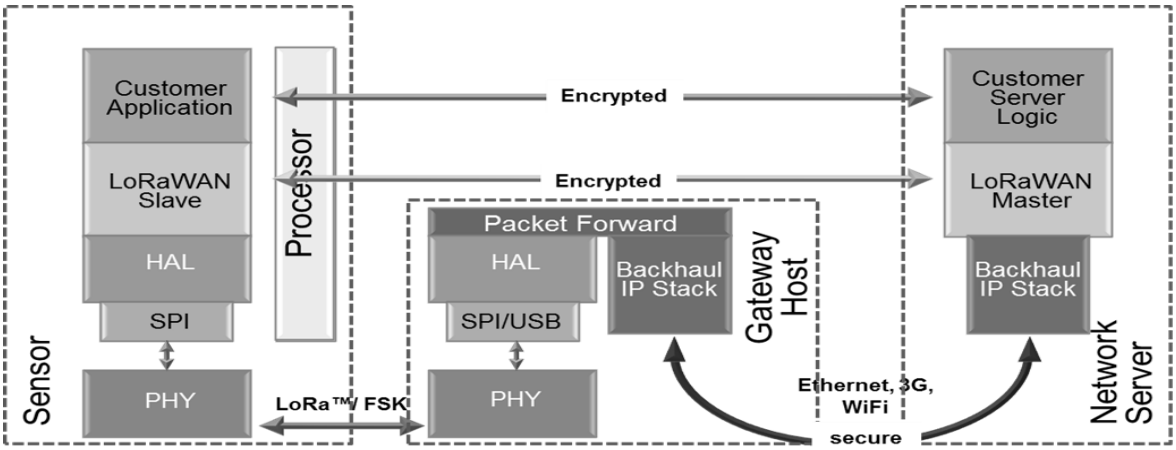

Figura 2.14: Arquitectura del protocolo LoRa
Fuente: LoRa Alliance

Seguridad

Una de las características más importantes de LoRAWAN es que tiene integradas medidas de seguridad, concretamente el cifrado y la firma de paquetes usando claves simétricas entre el gateway y los nodos. El intercambio de claves varía en función de cómo se una el nodo a la red. LoRaWAN ofrece dos formas de realizar este procedimiento:

Over The Air Activation (OTAA) es un método que requiere que el nodo tenga solo tres parámetros:

1. DevEUI: Número único globalmente que identifica al dispositivo (64 bits).
2. AppEUI: Identificador de aplicación que el gateway conoce previamente.
3. AppKey: Clave de aplicación que el gateway también conoce previamente.

El nodo envía un mensaje *join-request* con estos tres parámetros al gateway. Éste, si los parámetros son correctos, calculará las claves necesarias para la comunicación: Network

Session Key (NwkSKey) y Application Session Key (AppSKey). Creará un mensaje *join-accept* encriptado con AppKey, que incluirá la dirección que deberá tomar el nodo dentro de la red junto con algunos parámetros de configuración. Cuando el nodo reciba este mensaje podrá calcular independientemente las dos claves mencionadas anteriormente, que servirán para encriptar todos los futuros mensajes intercambiados.

Activation By Personalization (ABP) es un método alternativo en el que no se realiza el intercambio de claves que se lleva a cabo con OTAA, sino que DevAddr, NwkSKey y AppSKey están precalculadas e introducidas en ambos extremos de la comunicación. De esta forma no es necesario el procedimiento anterior y los nodos pueden comunicarse con el gateway directamente, sin necesidad de ningún proceso de configuración.

Es importante tener en cuenta, si se utiliza este último método, que las claves NwkSKey y AppSKey deberían ser generadas aleatoriamente, y no derivadas de información públicamente disponible como la localización de un nodo o su dirección MAC, ya que cualquier atacante podría hacerse con esta información y deducir las claves para suplantar al nodo y enviar mensajes en su nombre.

Gracias a estas medidas de seguridad se consigue, no sólo la autenticación de dispositivos, y por tanto el bloqueo de nodos no autorizados dentro de la red, sino el cifrado de todos los mensajes enviados entre los nodos y el gateway.

Por qué se ha elegido

LoRaWAN se ha elegido por presentar una serie de ventajas sobre otras soluciones con importancia crítica para el planteamiento inicial de este proyecto.

Escalabilidad

En primer lugar, el modelo centralizado que propone, permite una escalabilidad muy necesaria para este proyecto, reduciendo el proceso de crecimiento de la red a la simple

instalación de nuevos dispositivos en sus respectivas localizaciones. Todo el proceso de autenticación y conexión a la red LoRa se hace automáticamente siempre y cuando los dispositivos tengan acceso al Dev_EUI y la clave establecidos en el gateway. De esta forma, no es necesario ningún proceso manual para añadir nodos a la red mas que el de cargar el archivo de configuración relevante en cada uno, evitando así tener que acceder al gateway o que permitir manualmente la conexión.

LoRaWAN está diseñado para permitir que muchos dispositivos puedan conectarse a la vez a un solo gateway, pero hay un límite a este número que depende de las especificaciones del hardware utilizado. En ese caso será necesario instalar un segundo gateway para dar servicio a la misma red, y sincronizarlo con el primero para que no se procesen los mismos paquetes dos veces. En cualquier caso, por falta de dispositivos, no ha sido posible probar la estabilidad del gateway utilizado cuando un número elevado de nodos intentan comunicarse con él al mismo tiempo. Por esta razón no podemos siquiera estimar cuántos gateways habría que introducir en la red para dar servicio a un campus de universidad.

Independencia de internet

Otra gran ventaja que presenta el protocolo LoRaWAN frente a una solución más básica, como conectar los dispositivos al servidor directamente por WiFi o una conexión por cable, es que es completamente independiente de Internet. Esto es clave para un entorno tan activo como el del campus de una universidad, en el que las frecuentes fluctuaciones en el uso de las redes WiFi internas causan numerosos problemas que tienden a hacer decrecer su fiabilidad con el tiempo. Ni siquiera la conexión a la red cableada sería una solución óptima, puesto que complicaría enormemente el despliegue y obligaría a poner cables conectados a la red lo suficientemente largos en todos los puntos en los que se quiera colocar un nodo.

Estabilidad

Además, una red separada, específica para este propósito, no sólo tiene la ventaja de ser independiente de los frecuentes fallos de las redes públicas, sino que además estará mucho

menos congestionada, y tendrá un mayor grado de fiabilidad. Esto último se verá acrecentado cuando se pueda probar que la infraestructura se muestra robusta bajo distintos casos de uso, ya que estos presentarán patrones fácilmente estudiables, con un número de usuarios relativamente estable en función de la hora del día.

Seguridad

Otra ventaja de LoRa, quizá la más práctica de todas en lo que al desarrollo se refiere es el soporte nativo de los protocolos de seguridad detallados anteriormente, evitando así la preocupación de tener que implementar manualmente la seguridad de la comunicación, con todos los potencialmente desastrosos problemas que puede generar si no se hace correctamente. Gracias a esta característica ya implementada en el protocolo, la autenticación de dispositivos se reduce a introducir en un archivo de configuración las tres claves necesarias, y se delegan todos los procesos de intercambio de claves y autenticación a la implementación interna de LoRaWAN.

Alcance

Por último, el protocolo LoRaWAN tiene como una de sus grandes ventajas la característica que le da su nombre: el largo alcance. Gracias a tener un alcance teórico de entre 15 y 20 km, se puede hacer un despliegue en todo el campus de la universidad con un solo gateway central que dé servicio a todos los dispositivos. De otra manera serían necesarios varios gateways en varios puntos del campus para poder recibir correctamente las señales de cada edificio. Con esta solución, solo será necesario añadir más gateways a la red si el número de nodos es tan elevado que el hardware de red de dicho gateway alcanza su límite de conexiones simultáneas y se necesita instalar uno auxiliar para repartir la carga.

2.3.2. Módulo LoRa

Para la implementación de LoRa en este proyecto se ha usado un Microchip RN2483. Este dispositivo se conecta a la Raspberry a través de un socket XBee usando el protocolo

UART.

UART (*Universal Asynchronous Receiver/Transmitter*) es un protocolo muy simple de transmisión asíncrona de datos. Estableciendo unos parámetros comunes entre el emisor y el receptor, los bits datos se envían secuencialmente, con la posibilidad de separarlos en palabras, añadiendo un bit especial al inicio y otro al final de cada una que actúan como delimitadores. Opcionalmente se puede incluir un tercer bit de paridad para realizar un control de errores básico.

Esta división de los datos en palabras convierte a UART en un protocolo ideal para la transmisión de comandos de un dispositivo a otro, como por ejemplo en una terminal. En este caso, el dispositivo acepta una serie de comandos para configurar los parámetros del protocolo LoRa, por lo que para utilizarlo sería necesario usar algún programa de comunicación por puerto serie e introducir manualmente los comandos cada vez. Sin embargo, en este proyecto se ha desarrollado un programa específico usando las bibliotecas de C++ arduPi y arduPiUART, desarrolladas por CookingHacks.

Aunque el lector de NFC no utiliza UART, sino que se comunica con la Raspberry por SPI, el conectar dos dispositivos al mismo tiempo puede suponer, y ha supuesto, un conflicto en el uso de ciertos pines, ya que ambas placas pueden necesitar leer o escribir de un mismo pin al mismo tiempo. En este caso ambas dependen de un mismo pin, concretamente el GPIO23, lo cual ha generado un conflicto.

Para solucionar este problema, ha sido necesario modificar ligeramente la la configuración, para hacer que el módulo LoRa no dependa de este pin. Para ello, el pin del módulo que antes estaba conectado al GPIO23 se conectó a uno de alimentación a 3,3 voltios, y se modificó ligeramente la biblioteca de arduPi, como está detallado en el apéndice de instrucciones de uso.

A continuación se incluye un diagrama con el funcionamiento básico del programa desarrollado para la comunicación LoRa. Es importante notar que, el nodo no solo se reinicia cuando encuentra un error de conexión, también lo hará cuando haya reintentado realizar

una misma acción varias veces, como por ejemplo enviar un mismo paquete o conectarse al gateway.

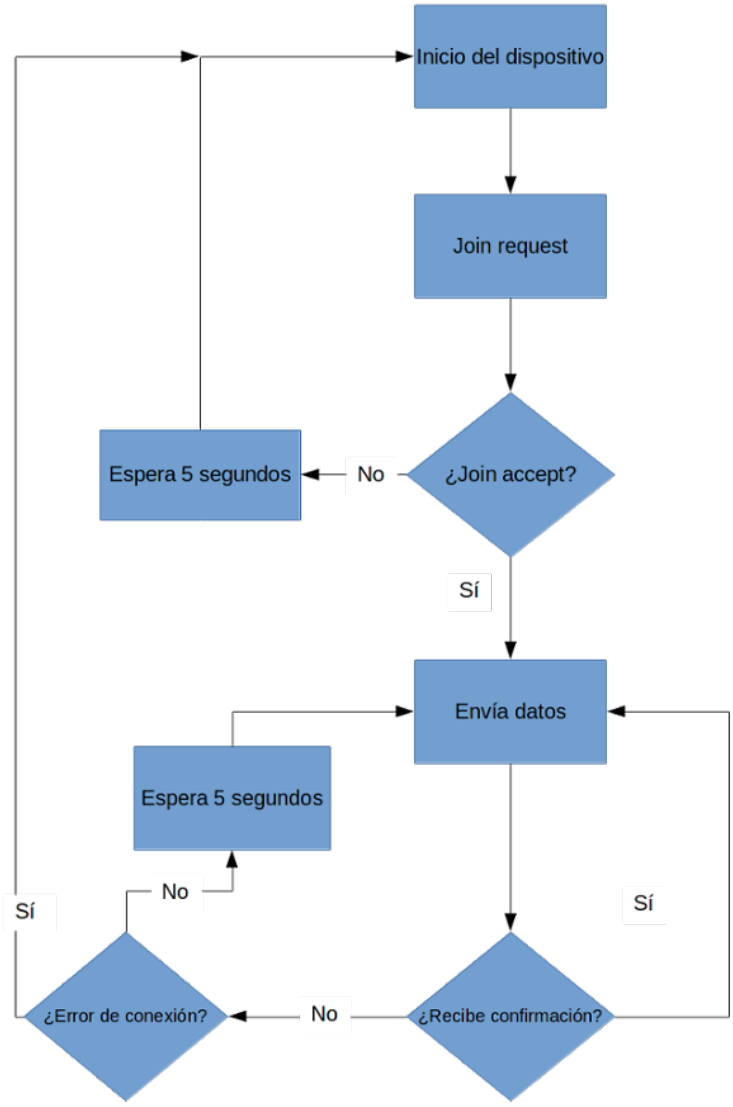

Figura 2.15: Diagrama del programa diseñado para LoRa

Capítulo 3

Gateway LoRa

3.1. Descripción

Una vez que el lector de NFC ha leído la información externa y se la ha transmitido al módulo de comunicación LoRa, el gateway tiene que poder recibir y procesar esta información. El modelo de gateway elegido para este proyecto es el Multitech Conduit, versión AEP. Aunque pueda parecer un modelo de coste elevado, tiene una serie de características que nos llevaron a tomar la decisión de adquirir uno de estos por encima de otros modelos:

Estabilidad

En concordancia con la información encontrada en diferentes sitios web, el hardware de red del gateway no ha demostrado ningún signo de inestabilidad en todo el tiempo que se ha estado utilizando y todas las pruebas que se han hecho. Era una característica importante que tener en cuenta a la hora de decidir un modelo, ya que la estabilidad de toda la infraestructura depende de que el gateway pueda mantenerse activo y dar servicio a todos los nodos. No obstante, si por alguna razón el gateway se mostrase inestable, se produjera un fallo de software o hardware o se desconectara intencionadamente por razones de mantenimiento, el software desarrollado para los nodos está preparado para almacenar temporalmente la información relevante y transmitirla cuando el gateway vuelva a estar operativo.

Capacidad

La documentación oficial del fabricante no parece indicar con precisión el número máximo de conexiones simultáneas que el dispositivo puede mantener activas sin sobrecargarse, y solo hace referencia a esta limitación esporádicamente, refiriéndose a una imprecisa cantidad de "miles de nodos". `REF` Debido al limitado número de nodos en nuestra posesión tampoco hemos podido probar este límite, pero sería necesario estudiarlo cuidadosamente en el caso de querer hacer un despliegue de la infraestructura en todo el campus de una universidad, ya que la carga a la que llegaría a estar sometido un solo gateway (potencialmente una decena de miles) probablemente sería demasiado elevada para los recursos que tiene. En ese caso sería necesario introducir un segundo gateway en la red para poder repartir la carga, como se ha explicado anteriormente.

Configuración

La mayor parte de la configuración del gateway se puede hacer directamente a través del portal de configuración que también trae instalado. Este portal tiene una interfaz web muy completa que permite configurar casi todos los parámetros necesarios para el gateway, desde las interfaces de red y la accesibilidad del propio portal de configuración hasta las opciones de LoRa y las aplicaciones de Node-RED (explicado en detalle más abajo), pudiendo incluso actualizar el firmware del propio dispositivo sin necesidad de tener acceso físico a él. Este portal puede hacerse accesible con una dirección IP pública a todo internet, lo cual ha resultado muy útil durante el desarrollo del proyecto, ya que nos ha permitido establecer o cambiar los datos de configuración desde cualquier máquina remota.

En algunos casos específicos en los que la interfaz web no ha sido suficiente para la configuración deseada, estableciendo una IP pública y los permisos correspondientes también se puede acceder al gateway por SSH, para así poder manejar directamente todos los programas y ajustes necesarios. Es importante remarcar que esto solo ha sido necesario en un pequeño número de ocasiones durante el desarrollo del proyecto, y que la interfaz de

configuración web ha demostrado ser lo suficientemente completa como para cubrir la gran mayoría de las necesidades.

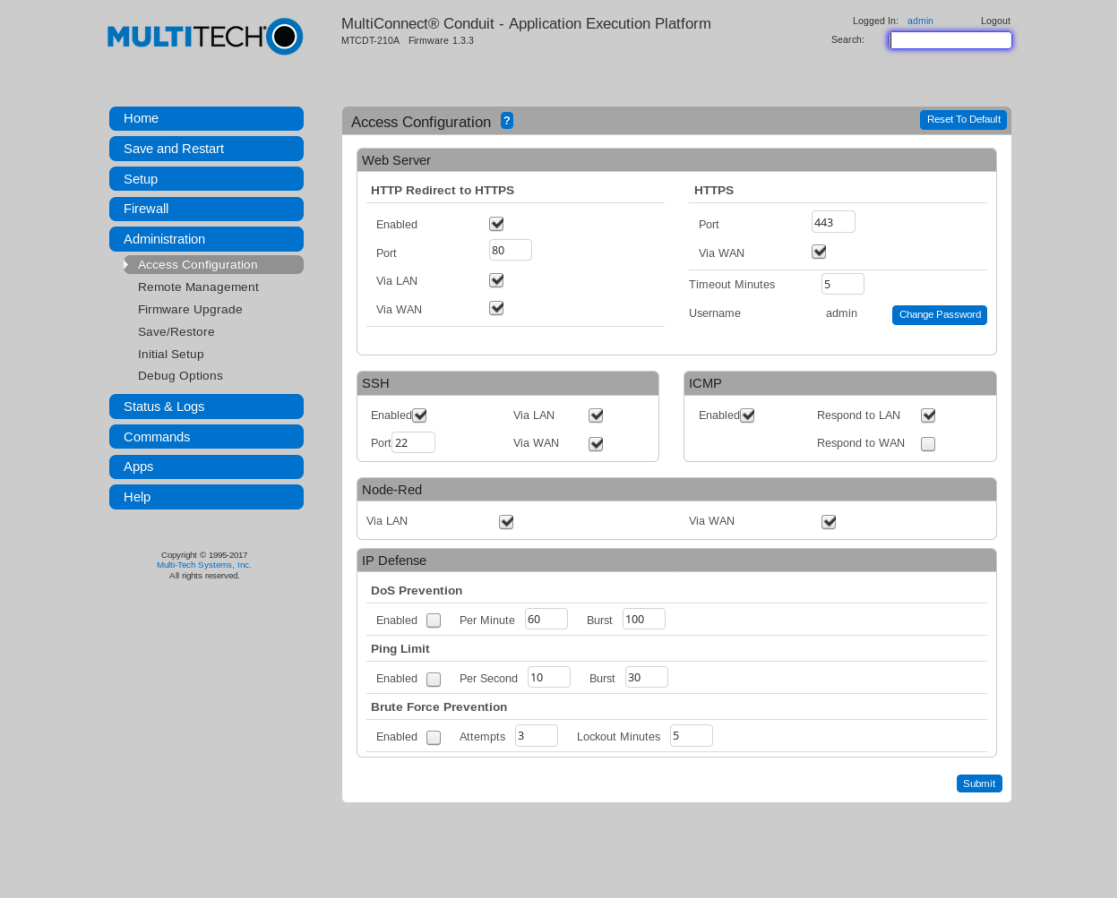

Figura 3.1: Portal de configuración del gateway

Software

El dispositivo trae una versión preinstalada de un entorno GNU/Linux, concretamente una distribución propia llamada mLinux, que está basada en la distribución minimalista Yocto Linux. Estas distribuciones están pensadas para sistemas con recursos reducidos como este, e incluyen las herramientas básicas que se pueden esperar en un entorno GNU/Linux, como un intérprete de shell, un editor de texto, un administrador de paquetes, openSSH o herramientas de configuración de redes y de control del sistema de ficheros. Los repositorios oficiales contienen software adicional como la máquina virtual de java o un cliente/broker

de MQTT (sobre los cuales se hablará más adelante), pero desafortunadamente no se ha encontrado ningún compilador nativo de C/C++, por lo que cualquier programa escrito por el usuario en estos lenguajes deberá ser compilado externamente con herramientas de compilación cruzada para ARMv5.

3.2. Node-RED

Quizá uno de los programas más importantes para este proyecto es Node-RED, una herramienta basada en el framework de Javascript, Node.js. Node-RED está pensada para simplificar lo máximo posible la conexión entre dispositivos hardware, APIs y servicios online. Proporciona un editor basado en el navegador que permite conectar de forma gráfica las distintas fuentes de información con diferentes puntos de salida. Cada bloque de Node-RED (llamados nodos), puede implementar un protocolo distinto, pero los datos que maneja dentro de Node-RED, ya sea generándolos o consumiéndolos, se encapsulan dentro de objetos de Javascript, lo cual facilita enormemente la tarea de procesarlos o modificarlos de cualquier forma. Algunos ejemplos de nodos presentes en Node-RED son MQTT, Email, TCP, Twitter, GPIO o Websocket, aunque las nuevas versiones de Node-RED permiten instalar de forma sencilla nuevos nodos desarrollados independientemente.

Figura 3.2: Node-RED

Además, Node-RED incluye algunos nodos auxiliares, que generalmente actúan de intermediarios entre nodos de entrada y de salida, y que permiten procesar o modificar la información recibida antes de enviarla a algún agente externo. Gracias a estos nodos auxiliares se pueden construir flujos de datos mucho más complejos, introduciendo retardos, diferenciación de caminos, formateo de datos a estándares como JSON o CSV, o funciones completas de Javascript que el usuario puede escribir para realizar el procesamiento que necesite. Adicionalmente, todos los nodos pueden incluir un pequeño menú de configuración para establecer algunos parámetros básicos relativos al protocolo o función a la que corresponden. La pestaña Ïnfo"suele incluir la documentación necesaria para configurar correctamente los nodos y saber cuál es el formato en el que esperan recibir los datos y qué tipo de información inyectan de nuevo en el flujo en caso de que lo hagan.

En particular los nodos correspondientes a LoRa (uno de entrada y uno de salida), que vienen preinstalados en el gateway han resultado de vital importancia para este proyecto, ya que el nodo de entrada de datos de LoRa es la única forma que hemos descubierto para obtener el contenido de los mensajes recibidos. Usando este nodo en Node-Red, es sencillo extraer los datos relevantes de cada mensaje LoRa que se recibe en el gateway, ya que esta plataforma funciona como un "demonio" y sus procedimientos se ejecutan constantemente. Esto permite que el gateway esté siempre pendiente de recibir información por LoRa para procesarla automáticamente como se haya especificado. En este caso, el flujo de Node-Red que se ha construido procesa la información decodificándola, guardándola en un archivo de log y enviándola por MQTT al servidor para que la almacene en la base de datos. Esto se consigue fácilmente en Node-RED simplemente seleccionando y arrastrando nodos y posteriormente estableciendo las conexiones entre ellos.

El flujo de Node-RED que utiliza el gateway para realizar el procesamiento descrito es este:

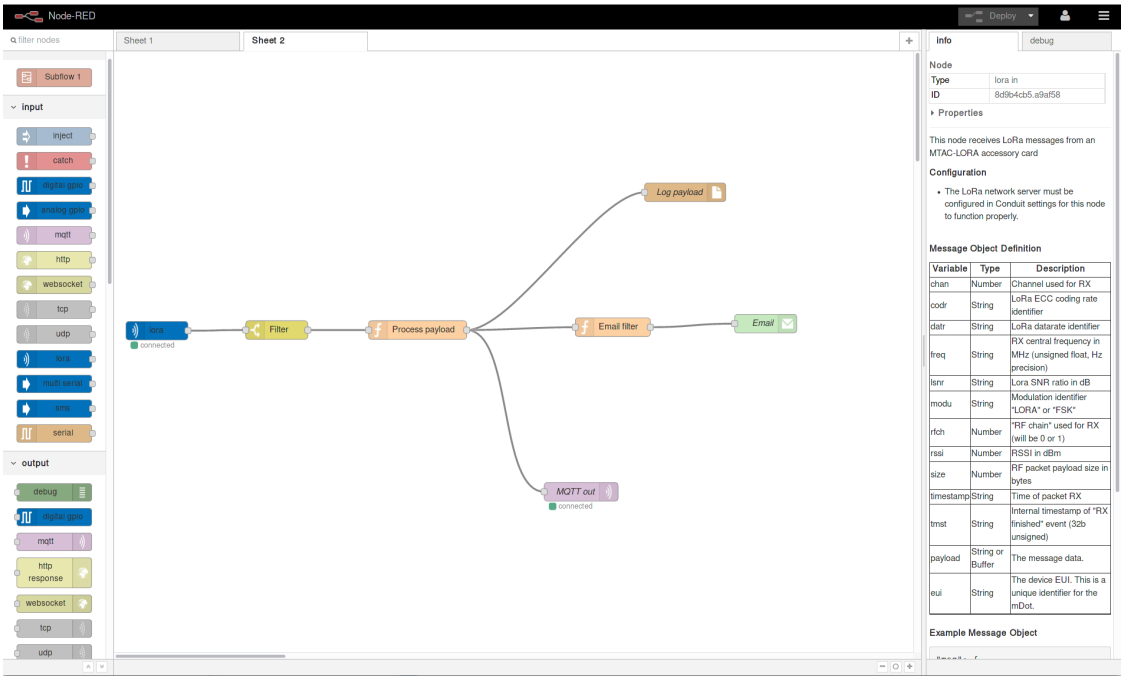

Figura 3.3: Flujo completo de Node-RED

3.3. MQTT

Cuando el gateway ha recibido los datos enviados por LoRa, su tarea será reenviarlos al servidor Kafka. Dadas las características del gateway, sería muy poco práctico utilizarlo como servidor al mismo tiempo, ya que con una capacidad de procesamiento y una memoria RAM tan limitadas, el sistema se vería sobrecargado con facilidad. Por esta razón se ha considerado más conveniente convertir al gateway en un simple receptor de paquetes de LoRa y delegar el resto de las tareas en otras máquinas.

Aunque la idea original era que el gateway enviase directamente los datos recibidos de LoRa a Kafka, se encontró una serie de problemas que obligaron a cambiar el planteamiento de esta comunicación. En primer lugar no se consiguió instalar el nodo de Kafka en Node-RED, debido a que la versión de Node-RED preinstalada en el dispositivo es demasiado antigua y no incluye el menú de instalación que tienen las versiones más modernas. Además, la arquitectura del procesador del gateway (ARMv5) dejó de estar soportada oficialmente por

Node.js, por lo que tampoco fue posible actualizarlo para poder instalar una nueva versión de Node-RED. Por otro lado, para poder ejecutar un cliente de Kafka, el gateway tendría que tener activa la máquina virtual de Java, pero los 256MB de RAM que tiene integrados no son suficientes para esto, por lo que se hace, a nuestro conocimiento, imposible que el gateway se comunique directamente por Kafka. Por esta razón se buscó una alternativa para establecer la comunicación entre el gateway y el servidor, para lo cual se consideraron varias opciones pero se acabó optando por MQTT. El protocolo MQTT, especialmente en su configuración más básica ha resultado el más sencillo de utilizar en este proyecto, solucionando el problema de la comunicación entre máquinas con la mayor rapidez. Aunque está diseñado para permitir arquitecturas de red arbitrariamente complejas con múltiples nodos, en este proyecto se utiliza para una sola conexión entre dos máquinas. Sin embargo presenta una serie de ventajas frente a una conexión manual con sockets a bajo nivel:

- Existen diversas implementaciones libres y accesibles para cualquiera que ya cumplen el objetivo que buscamos. Estas implementaciones incluyen ciertas funcionalidades de gran utilidad para este proyecto, como lanzar el broker en modo demonio para tener al servidor esperando datos constantemente.

- Viene integrado por defecto en Node-Red, por lo que no requiere ningún esfuerzo utilizarlo en una máquina tan limitada como el gateway. Desarrollar un programa para el gateway requeriría un proceso más complicado de lo normal, ya que no solo no tiene un compilador instalado localmente sino que además usa un procesador ARM, por lo que sería necesario utilizar herramientas de compilación cruzada en otras máquinas y luego transferir los ejecutables a su memoria interna.

- La especificación de MQTT incluye medidas de seguridad, que son de gran importancia para un sistema de red que se va a desplegar en un área pública como es el campus de una universidad. La primera es la restricción de acceso estableciendo un usuario y contraseña para asegurar que sólo los dispositivos autorizados pueden enviar y recibir

mensajes al servidor. La segunda es el uso de encriptación mediante SSL, dado que por defecto tanto el contenido de los mensajes como el usuario y la contraseña se envían en texto claro, y una simple inspección de paquetes en la red del servidor revelaría todos los datos instantáneamente.

- El protocolo MQTT incluye un parámetro llamado Quality of service o QoS, que indica si el receptor debe o no confirmar un mensaje cuando lo recibe. Si el emisor no recibe esta confirmación, asume que el mensaje se ha perdido y lo retransmite. Usando QoS 2, nos aseguramos de que todos los mensajes llegan a la base de datos una y solo una vez. Gracias a esto, deja de ser necesario preocuparse por los posibles fallos en la red, ya que el software que utilicemos se encargará de asegurar la integridad de los mensajes automáticamente.

Por estas razones, hemos decidido usar MQTT para la comunicación entre ambas máquinas. Usando el nodo que se incluye por defecto en Node-Red para enviar los mensajes desde el gateway y Mosquitto en el lado del servidor para hacer las funciones de broker y recibirlos, conseguimos que la información leída originalmente por el lector de tarjetas NFC llegue hasta el servidor. Una vez que el demonio de mosquitto ha recibido los mensajes, los almacena en un archivo en memoria para su posterior integración en la base de datos.

Capítulo 4

Servidor Kafka

En este capítulo hablaremos de qué es Kafka, y la funcionalidad que tiene en nuestro proyecto. La función principal de Kafka será la de recibir los mensajes enviados a través de la Raspberry tras haber pasado por el gateway LoRa, y en caso de que algún usuario desee que se le notifique que un concreto tópico ha recibido eventos, se le comunicará mediante una petición HTTP POST.

4.1. Apache Kafka

Apache Kafka[22] es un sistema de procesamiento de flujos de registros de código abierto desarrollado por Apache Software Foundation. Kafka fue desarrollado originalmente por LinkedIn y a comienzos de 2011 pasó a ser de código abierto. Kafka se basa en el modelo de mensajería por colas y editor-subscriptor. Tiene tres funciones principales:

1. Suscribirse a los streams (flujo de registros) y publicar en ellos.

2. Almacenar streams con una alta tolerancia a los fallos.

3. Procesar los streams mientras éstos se generan.

Kafka tiene cuatro APIs fundamentales:

- API Productor permite que una aplicación pueda enviar un flujo de registros a uno o más tópicos de Kafka.
- API Consumer permite que una aplicación pueda suscribirse a uno o más tópicos y procesar el flujo de los registros producidos con ellos.
- API Streams permite que una aplicación pueda procesar flujos de registros, de forma que consuma streams de entrada de uno o varios tópicos y generar con ellos streams de salida, almacenándolos de nuevo en Kafka o enviándolos a la aplicación elegida.
- API Connect permite el desarrollo de productores y consumidores para el envío de los datos contenidos en los tópicos de Kafka de forma segura a la aplicación elegida, como una base de datos.

Figura 4.1: Apache Kafka

En Kafka la comunicación entre los clientes y los servidores se realiza a través del protocolo TCP. La utilización de este protocolo nos facilita la compatibilidad entre versiones anteriores de Kafka.

Los servidores de Kafka requieren de ZooKeeper para su funcionamiento, (está incluido en el paquete de Kafka), por lo que antes de arrancar los servidores de Kafka, es preciso levantar el servicio de ZooKeeper.

ZooKeeper[33] es un servicio utilizado para el mantenimiento de la información de configuración, permitiendo la sincronización distribuida, permitiendo que los procesos se puedan

coordinar entre sí.

4.1.1. Tópicos

Un **tópico** es una categoría o tema en la cual se publican los registros. Los tópicos tienen un número indefinido de consumidores/productores, es decir, puede tener cero, uno o varios consumidores/productores que podrán leer los registros publicados en el tópico o publicar en ello.

Para cada tópico creado en Kafka, se mantiene un log particionado. Cada una de estas particiones es una secuencia ordenada de los registros que ha ido recibiendo Kafka. Cada uno de los registros tiene un id secuencial llamado desplazamiento (offset) que identifica a cada registro.

Kafka mantiene todos los registros, hayan sido o no consumidos, utilizando un período de retención configurable. Por ejemplo, si la política de retención se establece en un día, si publicamos un registro, éste estará disponible para ser consumido durante el periodo especificado, para posteriormente ser eliminado para liberar espacio. El rendimiento de Kafka no se ve afectado aun manteniendo un alto número de datos, por lo que el almacenamiento de datos durante un largo periodo de tiempo no tiene por qué provocar problemas a la herramienta.

De hecho, los únicos metadatos que se mantienen de cada consumidor es el offset o la posición del consumidor en el log. El offset o desplazamiento está controlado por el consumidor: normalmente avanzará su offset linealmente a medida que va leyendo los registros, pero, como controla la posición del offset, puede consumir los registros en el orden que desee.

4.1.2. Consumidores

Los **consumidores** se autoetiquetan con un nombre de grupo y cada registro publicado en los tópicos se entrega a una instancia del consumidor, y a cada uno de los consumidores suscritos al grupo. Si todos los consumidores pertenecen al mismo grupo de consumidores,

los registros repartirán la carga sobre los consumidores existentes. En el caso contrario, si todos los consumidores pertenecen a diferentes grupos de consumidores, cada registro se transmitirá a todos los procesos del consumidor.

4.1.3. Productores

Los **productores** publican los datos en los tópicos elegidos. El productor es quien se encarga de elegir qué registro se asignará a cada partición del tópico. Esto se puede hacer mediante Round-Robin para equilibrar la carga.

4.1.4. Ventajas

Gracias a la posible partición de los tópicos, esas particiones pueden ser replicadas, garantizando la tolerancia a fallos. Respecto a los grupos de consumidores, si para cada partición hay un grupo diferente de consumidores suscrito, los registros se van consumiendo por diferentes consumidores, garantizado que se mantiene el orden de entrega de los mensajes, mientras que el sistema se sigue balanceando.

4.2. Instalación y configuración

La versión de Kafka actualmente instalada en la máquina es la 0.10.1.0. Para su correcta instalación, y que la máquina pueda recibir eventos desde cualquier punto, además de poder enviarlos, es necesario modificar el fichero de configuración del servidor (server.properties), añadiendo la siguiente opción: advertised.host.name= <IP de la máquina>

Con este campo fijamos el nombre del host que se dará a los productores, consumidores y otros brokers para conectarse correctamente. Tuvimos que activar ese campo (por defecto está comentado) ya que, al realizar pruebas, Kafka era capaz de enviar y recibir eventos

desde la propia máquina, pero en caso de que tuviera que enviarlos a otra IP, o recibirlos, no era posible, ya que no anunciaba su IP de forma correcta.

4.3. Funcionalidad

Tras haber hablado de Kafka y su instalación en la máquina utilizada, pasamos a hablar de su utilización en nuestro proyecto. El primer paso es reenviar los datos recibidos a través de MQTT a Kafka, generando una especie de 'puente' entre ambas tecnologías. Este puente[35] consiste en un cliente MQTT que envía la información a un productor de Kafka, el cual reenviará esos mensajes a nuestro Kafka. En los primeros intentos de implementación, no conseguíamos que el cliente MQTT se conectara, por lo que desarrollamos un Conector a Kafka mediante su API (Kafka Connector). El conector funcionaba correctamente, era capaz de enviar los mensajes de MQTT a Kafka, pero los enviaba en un formato diferente al original y todos al mismo tópico, a diferencia de lo que queríamos, enviar los mensajes exactamente igual que como se habían recibido, y al tópico al que se había enviado originalmente, por lo que volvimos de nuevo a la idea del puente inicial.

El fallo en la conexión del cliente MQTT se encontraba en que no le habíamos dado los permisos necesarios para realizar la conexión, por lo que, una vez resuelto esto, ya fue posible reenviar los eventos recibidos en MQTT a Kafka.

Tras la conexión entre MQTT a Kafka, pasamos a la detección de eventos y su notificación. Los usuarios pueden suscribirse a los diferentes tópicos donde se almacena la información recibida en caso de que deseen que se les notifigue que un tópico ha recibido un evento o registro. Estos usuarios se suscriben almacenando en la **base de datos** la dirección URL en la que desean que reciban la notificación, y el tópico al que desea suscribirse. La base de datos consta de una tabla denominada suscripciones, que posee dos campos: *event*, que será el tópico al que se desea suscribir; y *user*, que será la URL a la que se quiere

enviar una petición POST en caso de que se detecte un evento en el tópico especificado en el anterior campo. Los usuarios pueden suscribirse a todos los tópicos que deseen, siempre que estos hayan sido creados previamente en Kafka.

Para realizar esto, hemos desarrollado un proyecto Maven[32] (Java) cuya función es generar un consumidor de Kafka, que esté suscrito a todos los tópicos existentes en Kafka. Este consumidor se mantiene activo siempre, a la espera de recibir eventos que consumir. En caso de que el consumidor detecte un evento, se realiza una consulta en la base de datos creada de forma que nos devolverá las URLS a las que se deberá hacer POST para la notificación.

Al realizar la petición POST, hemos incluido en el cuerpo de la petición un parámetro denominado 'topicPost', que contendrá el tópico que ha recibido un evento.

Es necesario que un servidor web esté levantado en todo momento para poder realizar las peticiones POST, ya que para poder crearlas, es preciso hacer una conexión HTTP a la URL a la que queremos realizar el POST. En concreto, hemos utilizado Apache Tomcat. Apache Tomcat[28] es un contenedor de Servlets desarrollado por la Apache Software Foundation. Tomcat implementa varias especificaciones de Java EE incluyendo Servlet Java, JavaServer Pages (JSP), Java EL y WebSocket, y proporciona un entorno de servidor web HTTP en el que se puede ejecutar código Java.

En la siguiente figura, podemos observar el flujo del proceso que hemos explicado en este proceso.

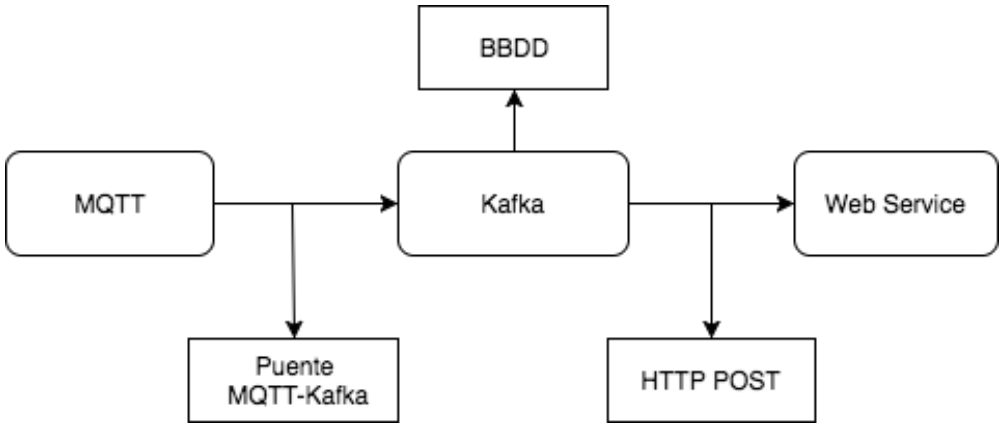

Figura 4.2: Flujo del proceso

Capítulo 5

WebService

Tras el capítulo de Kafka, en el que hemos contado que tras la detección de un evento, lo notificaremos a los usuarios suscritos mediante una petición HTTP POST, en este capítulo hablaremos de qué haremos con esas peticiones POST. Mediante un Web Service desarrollado con REST y JAX-RS, demostraremos que las peticiones se envían correctamente a este Web Service a través de un par de ejemplos.

5.1. REST

REST[31] significa REpresentational State Transfer, traducido al español, Transferencia de Estado Representacional, es un tipo de arquitectura que separa los elementos de esta arquitectura dentro de un sistema **hipermedia** distribuido.
La definición de hipermedia implica englobar todos los métodos o técnicas para diseñar, escribir o componer contenidos como audio, imágenes, texto plano, etc, de tal modo que el resultado de estas acciones, pueda interactuar con los usuarios. En REST, los datos y la funcionalidad se consideran los recursos y se puede acceder utilizando identificadores de recursos uniformes (URI), conocido normalmente como enlaces Web.

5.1.1. Principios

Para que un sistema pueda realmente definirse como REST, debe seguir una serie de principios de diseño:

- **Protocolo cliente/servidor sin estado:** Cada petición HTTP contiene toda la información necesaria para ejecutarse correctamente. Por tanto, permite que ni cliente ni servidor precisen de mantener ninguna información o estado previo para satisfacer la petición, aunque en la realidad, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión.

- **Operaciones bien definidas:** Se aplican a todos los recursos de información: HTTP define un conjunto de operaciones, de las cuales, las más importantes son: POST, GET, PUT y DELETE.

- **Uso de hipermedios:** Se utiliza para la información de la aplicación y sus transiciones de estado. Este estado del sistema suele representarse en un formato HTML o XML. El concepto de hipermedia explica la capacidad de una interfaz de desarrollo de proporcionar al cliente y al usuario los enlaces adecuados para realizar acciones concretas sobre los datos. Cualquier API REST debe cumplir el principio HATEOAS(Hypermedia As The Engine Of Application State - Hipermedia Como Motor del Estado de la Aplicación) para ser considerada una API REST.

- **Sistema de capas:** Cada una de las capas de las que se compone el sistema posee una funcionalidad diferente. Nuestro sistema no debe obligar al cliente a saber por qué capas circula la información, manteniendo la independencia del cliente fuera del sistema.

5.2. JAX-RS

Para el desarrollo del Web Service hemos utilizado la API de Java JAX-RS. La API JAX-RS[29] utiliza anotaciones de Java para simplificar el desarrollo de servicios web RESTful. Estas anotaciones JAX-RS se utilizan para definir los recursos y las acciones que se pueden realizar en esos recursos. Dentro de la arquitectura REST, se definen como **recursos** los datos y las funcionalidades que podemos encontrar en nuestro Web Service. JAX-RS se

sirve de anotaciones que durante el tiempo de ejecución, generarán las clases necesarias y los artefactos que precise el recurso, configurándolo de la forma deseada. Algunas de las anotaciones más utilizadas en JAX-RS son:

- **@Path:** Su valor es la ruta URI relativa que indica donde se encuentra alojada la clase Java: por ejemplo, /tfg. También se pueden incluir variables en la URI para hacer una plantilla de la ruta URI. Por ejemplo, se podría pedir el nombre de un tópico y pasarlo a la aplicación como una variable en la URI: /tfg/tópico.

- **@GET, @POST, @PUT, @DELETE, @HEAD** Se correspondería con la petición HTTP GET, HTTP POST, HTTP PUT, HTTP DELETE, HTTP HEAD (respectivamente) y procesa las solicitudes que reciba de este tipo.

- **@PathParam:** Esta anotación permite que se puedan extraer parámetros de la petición recibida para su uso en la clase donde se genere el recurso. Los nombres de los parámetros concuerdan con los nombres de las variables de la plantilla de la ruta URI especificadas en la anotación.

- **@QueryParam:** Esta anotación indica que se puede extraer un parámetro que podrá usar en las clases donde se generen los recursos.

- **@Consumes:** Se utiliza para especificar los tipos de representaciones media (XML, texto plano o HTML) enviados por el cliente que el recurso podrá consumir. Si @Consumes se aplica a toda la clase, todos los métodos de respuesta aceptarán los tipos de media especificados por defecto. Si se aplica a nivel de método, @Consumes sobreescribe cualquier anotación de ese mismo tipo aplicada a toda la clase.

- **@Produces:** Se utiliza para especificar los tipos de representaciones media que el recurso podrá generar y volver a enviar al cliente.

- **@Provider:** Esta anotación se utiliza para la obtención/lectura de datos en tiempo de ejecución. Para las peticiones HTTP, el MessageBodyReader se utiliza para asignar

el cuerpo de la petición a los parámetros de la función. En el caso de la respuesta, utilizamos MessageBodyWriter para añadir el valor de un parámetro al cuerpo de la respuesta que deseemos devolver.

5.3. Funcionalidad

El propósito principal de crear un Web Service es demostrar el funcionamiento de todas las partes anteriores del proceso, además de permitir que se pueda añadir mayor funcionalidad al proyecto en un futuro generando más servicios dentro del Web Service.

En este Web Service hemos programado dos servicios sencillos, que responderán a las peticiones POST que han sido enviadas al detectar que ha llegado un evento a Kafka, como hemos contado en el anterior capítulo. Cada uno de estos servicios, devolverá su respuesta al POST en dos formatos diferentes, XML y JSON.

De las anotaciones anteriormente citadas, hemos utilizado las siguientes:

- POST: Para indicar que la petición que requiere un servicio es de tipo HTTP POST.
- Produces: Para poder generar el contenido que se podría reenviar al usuario que realiza la petición. En el caso de XML, la anotación @Produces(“application/xml"), y para JSON, Produces(“application/json").
- Consumes: Al generar la petición POST, incluimos en el cuerpo un parámetro denominado "topicPost", parámetro en el que indicamos qué tópico ha recibido un evento. Para poder utilizar ese parámetro en el servicio, es necesario utilizar esta anotación de la siguiente forma: @Consumes(“application/x-www-form-urlencoded").

Dado que el nodo aún no ha sido configurado de forma que pueda recibir la respuesta a las peticiones POST que ha realizado, usaremos *Postman*, para ver qué información devolvería cada servicio a las peticiones POST realizadas.

Postman es una herramienta que nos permite construir y gestionar peticiones a servicios REST (POST,GET,etc). Su utilización no tiene ninguna dificultad ya que simplemente

tenemos que definir la petición que queremos realizar.

Al enviar la petición desea, Postman queda a la espera de la respuesta a la petición enviada. La herramienta es capaz de capturar las respuestas y mostrarnos el resultado de una forma clara y ordenada, ya sea en formato XML, JSON o Texto

Figura 5.1: Postman

En las siguientes figuras hemos simulado el comportamiento que tendrían las peticiones POST enviadas desde Kafka, como hemos indicado en el anterior capítulo. Para ello, hemos utilizado dos direcciones URL que se encuentran en la base de datos utilizada para Kafka y, para el parámetro 'topicPost', el tópico asociado a esas URLs. Las URLs utilizadas son:

`http://localhost:8080/WebserviceTFG/tfg/JSON`

`http://localhost:8080/WebserviceTFG/tfg/XML`

- **WebserviceTFG/tfg:** Indica la ruta del Web Service al que queremos realizar la petición.
- **JSON, XML:** Indica el servicio del Web Service al que queremos realizar la petición, JSON para devolver la respuesta del POST en formato JSON, y XML, para devolver la respuesta al POST en formato XML.

El Web Service recibirá las peticiones y, en función de la URL, el Web Service ejecutará el servicio elegido y devolverá su respuesta, en esta caso, a Postman.

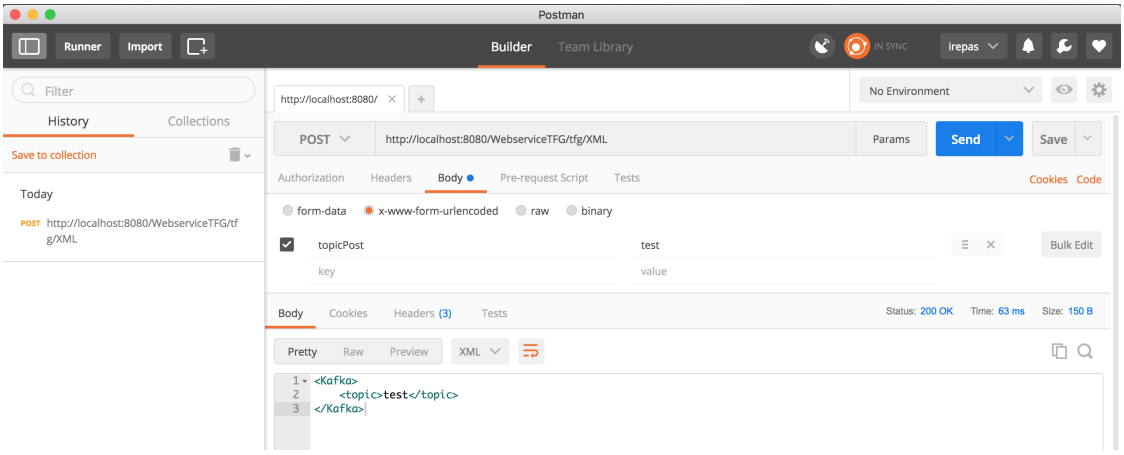

Figura 5.2: Respuesta POST formato XML

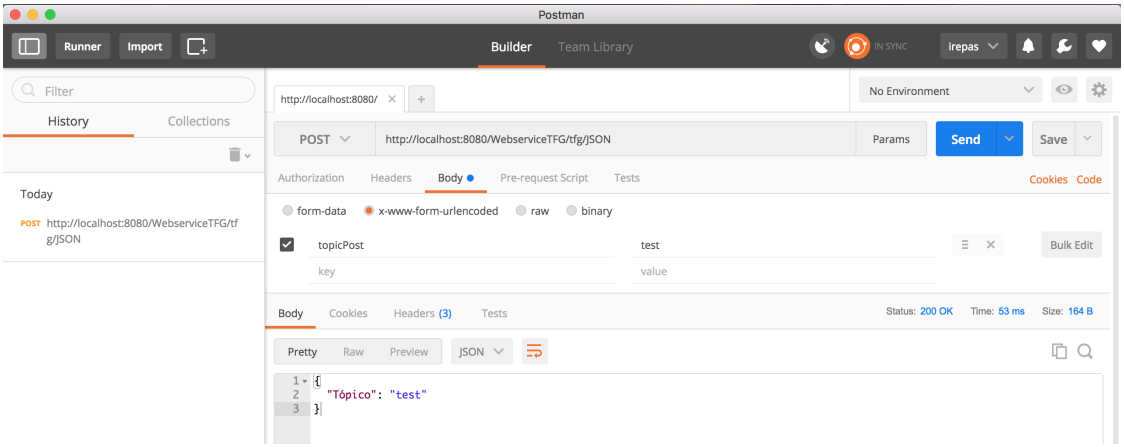

Figura 5.3: Respuesta POST formato JSON

Capítulo 6

Conclusiones

Una vez terminado este proyecto, se puede decir que hemos cumplido los objetivos propuestos, aunque, considerando el desarrollo de los últimos meses, es inevitable notar que estos objetivos han cambiado con bastante frecuencia. Sin intención de explicar con el máximo detalle todos estos cambios, que es objetivo de otras secciones de este documento, el planteamiento original que incluía NFC, LoRa y Kafka ha sido modificado en múltiples ocasiones, según las circunstancias lo han requerido.

En algunos casos los cambios fueron introducidos debido a que se encontraron limitaciones inesperadas del hardware, como es el caso del gateway que nos obligó a introducir MQTT en el flujo de datos, y en otros por no ser capaces de configurar o utilizar correctamente un dispositivo, como por ejemplo el módulo LoRa de Dragino que tuvo que ser reemplazado por la combinación de *shield* de Arduino y Microchip RN2843, con el coste añadido que esto supone.

Además, algunas de las ideas originales relativas a la lectura de las TUI tuvieron que ser descartadas, ya que no fuimos capaces de leer cierta información de las tarjetas que se encuentra en bloques actualmente inaccesibles de su memoria interna, debido a que la API utilizada aún no tiene implementada esta funcionalidad. Este es otro ejemplo de cambio que fue necesario realizar al planteamiento original, pero es un problema que no pudimos prever y con el que tuvimos que lidiar inevitablemente cuando el proyecto ya estaba en desarrollo. Finalmente no hubo otra opción que abandonar parte de la idea original, que incluía guardar

en cada tarjeta un contador de cuántas veces había sido leída, para así poder dificultar en cierta medida el fraude por duplicado de tarjetas.

Con estos y otros problemas que surgieron durante el desarrollo, nos vimos obligados a cambiar en algunos casos la forma de la implementación, y en otros la aplicación final del proyecto, con impacto directo en su posible uso final. Sin embargo, observando el resultado obtenido al finalizar el desarrollo, estamos satisfechos de cómo ha terminado. Cumple en gran medida los objetivos que se propusieron, y los problemas encontrados, aunque abundantes, no han impedido que el proyecto salga adelante y realice el propósito general que se propuso.

Dado que ninguno de los tres alumnos tenía experiencia previa en el desarrollo de un proyecto similar, este proceso ha supuesto dedicar muchas horas de investigación sobre las tecnologías y los dispositivos utilizados, y por ello una experiencia de aprendizaje sobre muchos temas. Y no sólo un aprendizaje técnico sobre estos protocolos o el *Internet of things*, sino de la metodología de trabajo que implica un proyecto de este tipo, lo que significa trabajar con otras personas durante todo un año, la importancia de planear y de leer detenidamente la documentación necesaria, y sobre todo la paciencia y la capacidad de sobreponerse a los errores y saber buscar otras soluciones cuando algo no funciona. Por todo esto, los tres podemos estar de acuerdo en que ha sido una experiencia gratificante en la que hemos aprendido muchas cosas.

6.1. Trabajo futuro

A continuación se presentan algunas ideas para mejorar el proyecto, tal vez en un trabajo futuro. Algunas son ideas que han quedado fuera del planteamiento original y otras simplemente no han podido llevarse a cabo por alguna dificultad encontrada durante el desarrollo:

Reemplazar el módulo LoRa por uno más barato. Aunque esto suponga replantear completamente el funcionamiento del nodo LoRa, el coste de la solución actual es demasiado elevado para un despliegue a gran escala. Una opción que se consideró durante el

desarrollo del proyecto fue el LoRaHAT de Dragino, pero acabó descartándose cuando el actual demostró ser más sencillo de configurar y usar.

Hacer reversible la comunicación . Actualmente la transmisión de datos solo funciona en un sentido, es decir, desde la Rasbperry hasta el Webservice. Tal y como está construida la infraestructura no supondría mucho trabajo hacer que la información fluya en sentido contrario.

Añadir un contador a las TUI . Como ya se ha comentado, la biblioteca que se usa para leer la información de las tarjetas por NFC aún no incluye la funcionalidad necesaria para escribir en las tarjetas de la UCM, por lo que no se ha podido incluir esta idea en el proyecto. Añadir un contador de lecturas a las tarjetas permitiría detectar en algunos casos el fraude por duplicación.

Distinguir entre usuarios . Tomando el UID leído de la TUI, podría llevarse un registro de a qué tipo de usuario pertenece. Esta modificación, junto con la de hacer que la comunicación sea bidireccional es necesaria si se quiere cumplir el objetivo inicial de realizar un control de acceso a ciertas zonas, permitiendo que sólo algunos usuarios puedan abrir las puertas usando su tarjeta.

Bibliografía

[1] *NFC - Información.* https://es.wikipedia.org/wiki/Near_field_communication.

[2] *LoRaWAN - Información.* https://es.wikipedia.org/wiki/LoRaWAN.

[3] *Python - Información.* https://en.wikipedia.org/wiki/Python_(programming_language).

[4] *Raspberry Pi - Documentacion.* https://www.raspberrypi.org/documentation/.

[5] *Raspbian - Información.* https://www.raspbian.org/RaspbianAbout.

[6] *GPIO - Pines.* https://es.pinout.xyz/.

[7] *SPI - SPI en Raspberry Pi.* https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md.

[8] *UART - UART en Raspberry Pi.* https://www.raspberrypi.org/documentation/configuration/uart.md.

[9] *Raspbian Jessie - Descarga.* https://www.raspberrypi.org/downloads/raspbian/.

[10] *PNEV512R - Documentación.* http://www.nxp.com/documents/application_note/AN11480.pdf.

[11] *Tarjeta TUI - Tecnología.* http://www.observatoriotui.com/evolucin-a-tui-r7/tui-r5r6.

[12] *MIFARE Classic 4K - Documentación.* http://www.nxp.com/documents/data_sheet/MF1S70YYX_V1.pdf.

[13] *MIFARE Classic 1k - Documentación.* http://cache.nxp.com/documents/data_sheet/MF1S50YYX_V1.pdf.

[14] *NFC Software - Descarga.* http://www.nxp.com/products/identification-and-security/nfc-and-reader-ics/nfc-frontend-solutions/explore-nfc-exclusive-from-element14:PNEV512R?tab=Design_Tools_Tab.

[15] *nxppy - Librería.* https://github.com/svvitale/nxppy.

[16] *NXP NFC Reader Library - Documentación.* http://www.nxp.com/documents/application_note/AN11802.pdf.

[17] *INA219 - DataSheet.* http://www.ti.com/lit/ds/symlink/ina219.pdf.

[18] *NodeMCU - Documentación.* https://nodemcu.readthedocs.io/en/dev/.

[19] *LoRa - Documentación.* https://www.lora-alliance.org

[20] *Gateway LoRa - Documentación.* http://www.multitech.net/developer/products/conduit/

[21] *MQTT - Documentación.* http://mqtt.org/documentation.

[22] Kafka - Documentación. https://kafka.apache.org/documentation/.

[23] *Mosquitto - Broker.* http://mosquitto.org/.

[24] *Mosca - Broker.* https://github.com/mcollina/mosca.

[25] *Node-Red.* *http://nodered.org/*.

[26] I^2C - Module. https://nodemcu.readthedocs.io/en/dev/en/modules/i2c/.

[27] *MQTT - Module.* *https://nodemcu.readthedocs.io/en/dev/en/modules/mqtt/*.

[28] *Apache Tomcat.* https://kafka.apache.org/intro.

[29] *JAX-RS - Documentación.* http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html.

[30] *Postman.* https://www.getpostman.com/.

[31] *REST - Documentación.* http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html.

[32] *Maven - Información.* https://maven.apache.org/.

[33] *Zookeeper - Información.* http://zookeeper.apache.org/.

[34] nxppy - Issues. https://github.com/svvitale/nxppy/issues/5.

[35] *Puente MQTT - Kafka.* https://github.com/accexar/mqttKafkaBridge.

Apéndice A

Contribución personal de cada integrante del grupo

A.1. Fátima

A lo largo de este proyecto, nuestras tareas han estado bastante diferenciadas y definidas. En mi caso, la mayor parte del desarrollo del proyecto la he invertido en el diseño del nodo lector. Mi tarea principal ha sido la construir un prototipo basado en un sistema Raspberry que fuera capaz de leer información de las tarjetas TUI a través del dispositivo EXPLORE-NFC integrado.

Mi trabajo comenzó con la configuración de la Raspberry Pi. Lo primero de todo era dejar preparado y funcionando el sistema. Se nos proporcionó una Raspberry Pi 3 Model B, por lo que lo primero que tuve que hacer fue documentarme sobre la instalación y configuración inicial de la misma. Para ello me ayudé de la documentación proporcionada a través de la web oficial de Raspberry Pi.

Una vez tenía claro los pasos a seguir, me hice con los componentes necesarios (microSD, teclado, ratón, monitor, cable HDMI y fuente de alimentación) y comencé con la instalación, la cual era bastante sencilla y sin complicaciones. Descargué Raspbian Jessie Lite y seguí los pasos, monté la imagen en la microSD y, con todo conectado, en unos minutos ya tenía

la Raspberry Pi en funcionamiento. Con el fin de manejarme de manera cómoda y rápida, activé el protocolo SSH y descargué como cliente PuTTy para poder trabajar con ella desde mi ordenador.

Lo siguiente, y lo que comenzó a dar algunas complicaciones fue la parte de NFC.

Por un lado, conectar la placa NFC PNEV512R y ponerla en funcionamiento no tuvo mucha complicación, aunque me llevó unos días hacerme con la documentación actualizada del dispositivo, ya que los manuales y las versiones de software variaban entre unos proveedores y otros. Finalmente, gracias a la documentación del módulo PNEV512R proporcionada por la web de NXP, en concreto el manual AN11480, conseguí poner en funcionamiento el dispositivo NFC conectado a la Raspberry Pi con el protocoolo SPI activo. Hasta entonces tenía los dos dispositivos integrados y funcionando.

Nota: Muchas veces, el protocolo SPI de la Raspberry Pi se desactiva, por lo que es importante revisarlo si EXPLORE-NFC deja de funcionar y volverlo activar.

Las complicaciones empezaron a la hora de desarrollar el código que leería las tarjetas TUI. Como primera opción elegí C como lenguaje de programación, puesto que NXP proporcionaba su propia librería NFC NXP Reader Library escrita C y que contaba con multitud de funcionalidad. Además, se trataba de un lenguaje que conocía y con el que he ido trabajando a lo largo de la carrera. Descargué la librería, también disponible en la web de NXP, y seguí sus instrucciones de uso además de documentarme sobre la misma. Sin embargo, pasaron semanas y aún no había conseguido hacer funcionar los códigos de prueba (compuestos por gran cantidad de líneas de código) proporcionados por la librería con el módulo EXPLORE-NFC. Al cabo del tiempo, a través de foros descubrí que la librería no era compatible con este dispositivo. Fue entonces cuando tomé como solución Python y su librería `nxppy`, lo que en C eran cientos de líneas de código, con Python disminuían considerablemente manteniendo la claridad del código.

Descargué la librería y comencé a realizar pruebas. Rápidamente conseguí leer el identificador de la tarjeta TUI y devolver la fecha y hora de lectura, incluso si el tipo de tarjeta

TUI era 1K ó 4K. A partir de aquí, se planteó la idea de añadir e inicializar un contador propio a la tarjeta, que aumentara en 1 cada vez que se realizara una lectura o escritura en ella. Era una forma sencilla de asegurarnos, en caso de duplicación de tarjetas, cuál sería la original. Conseguí fácilmente añadir dicho contador a la tarjeta de prueba MIFARE Classic 1K, simplemente elegí un bloque en el que almacenar dicha información y a partir de ahí leer y escribir en dicho bloque, simulando el contador.

Sin embargo, no se podía acceder tan fácilmente al contenido de las tarjetas universitarias de tipo MIFARE Classic 4k. Fue ahí cuando empecé a buscar información sobre éstas y, como se ha detallado en la sección 2.2.2, para poder leer o escribir contenido en ellas es necesario autenticarlas previamente, es decir, conocer una de sus dos claves. Tras muchos días leyendo documentación finalmente, gracias a los *issues*[34] abiertos en el proyecto de `nxppy` de Github, descubrí que la librería nxppy no incluía funcionalidad necesaria para autenticar las tarjetas de tipo 4K, aunque sí para las de 1K. Basándome en la funcionalidad proporcionada para autenticar las tarjetas MIFARE Classic 1K, intenté agregar dicha funcionalidad a la librería para las de tipo 4K, pero finalmente no conseguí hacer que funcionara. Es por eso que decidimos dejar cerrada la parte de lectura NFC y conformarnos con devolver el identificador de la tarjeta.

A continuación, dediqué unos días a Node-RED y fui probando y creando mis propios flujos ya que la idea original era comunicarse de esta manera con el cliente Kafka del servidor. Sin embargo, esta idea finalmente se descartó por problemas de versiones y optamos por MQTT como alternativa.

Por último, una de las tareas más importantes de este proyecto fue integrar todo nuestro trabajo. En el caso del nodo, teníamos una Raspberry con NFC funcionando y leyendo las tarjetas TUI; y otra con el cliente LoRa mandando los datos recogidos al gateway. Junto con Óscar, dedicamos unos días a conectar éstos dos dispositivos a una sola Raspberry con un extensor de pines GPIO. Nos encontramos con problemas ya que los dos dispositivos intentaban hacer uso del mismo bus SPI y por tanto, no trabajaban al mismo tiempo en la

misma Raspberry. Afortunadamente, Óscar consiguió hacerlos funcionar, dándose cuenta de que el módulo LoRa podía trabajar a través del puerto serie UART y finalmente, modificando la librería, se consiguió integrar dejando nuestro prototipo de nodo lector funcionando.

A.2. Óscar

Durante el desarrollo del proyecto me he encargado principalmente de todo lo relacionado con LoRa. He dedicado la mayor parte del tiempo a leer documentación de varias tecnologías y varias soluciones distintas a los problemas que hemos necesitado solucionar. Desde el inicio, el planteamiento en cuanto a cómo se establecería la comunicación entre los dispositivos fue bastante impreciso y se propusieron algunas soluciones que nunca llegaron a funcionar, lo cual supuso investigar aún más y buscar nuevas ideas para hacer funcionar la infraestructura.

Como ejemplificación de esto último, una de las ideas iniciales era usar un adaptador de XBee a USB para poder conectar el módulo LoRa a un ordenador y comunicarse con él por puerto serie. Con esto se pretendía montar un gateway básico y temporal hasta que se comprase el producto definitivo. Esto, con mis conocimientos de UART, LoRa y la comunicación por puerto serie del momento resultó una tarea mucho más complicada de lo planeado, y finalmente no resultó una solución viable, ya que el propio fabricante del dispositivo confirmó que no se podía hacer.

Como el desarrollo de la parte de LoRa tuvo que detenerse durante un tiempo debido a que el pedido del gateway tardó en realizarse, me dediqué a estudiar la documentación y el funcionamiento de los distintos módulos de LoRa que se plantearon para el nodo formado con la Raspberry.

Cuando se compró finalmente el gateway, dediqué una gran cantidad de tiempo a leer documentación sobre él, y a explorar las distintas opciones y funcionalidades que tienen el gateway y los otros dispositivos incluidos en el starter kit que fue adquirido. Los otros nunca llegaron a ser de utilidad fuera de realizar pruebas de LoRa con dispositivos distintos de la Raspberry, pero también dediqué tiempo a aprender a manejarlos.

Un asunto que resultó particularmente complicado fue el de obtener el contenido de los mensajes LoRa una vez que el nodo fue capaz de enviarlos y el gateway de recibirlos. Finalmente, después de mucho buscar, la única solución que fui capaz de encontrar fue

Node-RED, que a diferencia de todo lo demás, resultó ser enormemente sencillo de utilizar, y fui capaz de construir el flujo que se usaría definitivamente en la versión final del proyecto con relativa facilidad.

Sin embargo, la gran fuente de dificultades resultó ser el propio gateway. No por su hardware de red, que no ha ocasionado ningún problema por fallo o mal funcionamiento, sino por la limitación de sus recursos. Aunque, tener un sistema Linux preinstalado parecía bastante más de lo que se podía esperar de un dispositivo de este tipo, eso no significó que fuese lo suficientemente potente como para realizar siquiera las tareas más básicas. Dos de los principales problemas causados directamente por estas limitaciones fueron repercutieron directamente en el planteamiento del proyecto.

En primer lugar, al tener equipado un procesador fabricado en 2001, con una arquitectura tan antigua como ARMv5 no permitió actualizar Node.js ni, por extensión, Node-RED. Esto signficó que debimos mantener una versión desactualizada en la que no se puede (o no fui capaz) importar nuevos nodos. Esto obligó a cambiar la idea original del proyecto ya que pensábamos hacer que Node-RED se comunicase directamente con el cliente Kafka del servidor, pero como el nodo de Kafka no viene preinstalado en Node-RED y no fui capaz de incluirlo por tener una versión tan antigua, nos vimos obligados a introducir MQTT en el proyecto. Igualmente, aunque intenté crear un productor de Kafka independiente de Node-RED, el gateway no fue capaz de ejecutarlo porque los 256MB de RAM que tiene instalados no son suficientes para ejecutar la máquina virtual de java de la que depende Kafka. Por suerte MQTT sí está incluido en Node-RED de fábrica, no sobrecarga los recursos tan limitados del gateway y permitió que se pudiera establecer la conexión con el servidor de esta manera.

El segundo problema que causó el sistema del gateway fue que con la distribución mínima de Linux que tiene instalada, no fui capaz de conseguir utilizar MQTTs, de modo que la conexión se tiene que realizar mediante MQTT estándar, que no está encriptado. Después de muchas horas haciendo pruebas para intentar conseguirlo y mucha documentación leída

en internet, tuve que abandonar el intento de usar la versión segura de MQTT, aunque ha funcionado sin ningún problema en todos los otros dispositivos que he podido probar.

Finalmente, cuando todas las comunicaciones estuvieron establecidas, dediqué un tiempo, con la ayuda de Fátima, a conectar los dos dispositivos a la Raspberry al mismo tiempo, que resultó no ser una tarea fácil y supuso acabar por modificar la biblioteca que se utilizaba.

A.3. Irene

Durante el proyecto, me he encargado principalmente del enlace entre MQTT y Kafka y de todo el desarrollo asociado a Kafka y el Web Service. Para ir conociendo el funcionamiento de Kafka, primero instalé Kafka en mi ordenador y programé códigos sencillos de consumidores y productores para irme familiarizando con Kafka. Una vez los tenía funcionando en local, hice las mismas pruebas en la máquina que los tutores nos han dejado utilizar durante el proyecto. Una vez Kafka funcionaba correctamente en la máquina, el siguiente paso era probar la comunicación entre local y la máquina enviando eventos de local a máquina y viceversa.

Uno de los problemas con los que me encontré, es que no podía probar el funcionamiento entre local y la máquina en la Facultad de Informática dado que, al hacer las pruebas, comprobé que a través de redes públicas WiFi como eduroam y UCM, no es posible enviar eventos a una dirección IP que no fuera la local. Al principio pensaba que era por no indicar bien los parámetros al enviar los mensajes, como por ejemplo la IP de destino o la IP de ZooKeeper, pero al realizar las pruebas con una red privada, funcionó en el primer intento realizando las pruebas exactamente de la misma forma que en la facultad.

Por tanto, durante todo el desarrollo del proyecto, cada vez que he querido realizar pruebas que implicara comunicación entre máquina y local, he tenido que hacerlo en lugares con redes privadas.

Tras una reunión con el profesor Iván Martínez-Ortiz, en la que nos habló sobre los Webhooks de Github, planteamos imitar su comportamiento en Kafka, por lo que comencé a desarrollar el código que detectaría que un evento ha llegado a Kafka y el envío posterior de la petición HTTP POST para su notificación.

En esta parte del desarrollo del proyecto, lo que más problemas me dio fue escribir el código encargado de enviar las peticiones POST, principalmente por las librerías que tenía que utilizar para generar un POST. Primero probé con HTTP Client de Apache, pero me daba problemas al generar correctamente la URL a la que realizar el POST, por lo que

finalmente opté por las librerías ya incluidas en Java. También me dio problemas detectar los eventos y buscar la URL de forma simultánea, ya que la búsqueda en la base de datos de donde sacamos las URLs a las que se hacen las peticiones POST, era más lenta que la detección de los eventos, provocando que no se confirmara la detección de éstos y un mismo evento se detectaba como recibido al menos dos o tres veces. Esto se solucionó ejecutando la búsqueda en la base de datos en un proceso paralelo, en lugar de hacerlo en el mismo proceso que la detección de eventos.

Para realizar las primeras pruebas de que las peticiones POST se enviaban y llegaban correctamente, hice un pequeño proyecto JSP (Java Server Pages) que al recibir una petición POST, mostrara un mensaje de notificación. Para corroborar que las peticiones se realizan correctamente y que en un futuro se le pueda dar mayor funcionalidad, definiendo funciones concretas como las mencionadas en la introducción y el resumen, desarrollé un Web Service siguiendo la arquitectura REST que recibe peticiones POST y devuelve un mensaje en formato XML o JSON, en función de a qué dirección se ha realizado la petición.

Como el nodo no está preparado para recibir lo que las peticiones POST hayan devuelto como respuesta, encontré la herramienta *Postman*, que simula un cliente del Web Service.

La funcionalidad que se le he dado al Web Service es mínima, ya que el nodo no está configurado de forma que pueda recibir la información que le puede aportar el Web Service.

Dentro del desarrollo del Web Service no tuve muchas complicaciones, dado a que la arquitectura REST facilita bastante la definición de los servicios por el sistema de anotaciones que utiliza, y que los servicios desarrollados son muy simples como he dicho anteriormente.

Ya para finalizar el proyecto, era necesario que los datos que llegaban a MQTT, fueran reenviados a Kafka para pasar por todo lo descrito anteriormente. No tuvo apenas complejidad ya que en GitHub encontré desarrollado lo que necesitaba: un cliente MQTT que reenviara sus datos a un productor de Kafka, ya compilado en formato .jar y con las dependencias necesarias actualizadas.

Apéndice B

Introduction

The main idea of the project is to develop a system based on low cost nodes, in our case we have used Raspberry, that is able to read the students' card (TUI Cards) through NFC, receiving a series of data from the cards that can be used later for various applications.

One of the main utilities that could have the system is the control of attendance class without the need for teachers to waste time in this task. It could also be used to control access to University facilities, such as garages, sports facilities, laboratories, etc. The possibility of implementing our project in the university and the utilities that could contribute are the reasons that have motivated us to carry out this Project, because it would facilitate various aspects of daily life at the university for all who are part of it (students, teachers, administrative staff, etc.).

As our purpose is that this project can be implemented in the university, it would be necessary to install nodes at many points, but not all places where we would want implement the nodes, would have access to WiFi, so with respect to the scope of connectivity, we have chosen LoRa.

LoRa is a technology used to communicate small electronic devices used in the so-called Internet of Things (IoT). It is useful for low power networks and wide area. The read data will be sent through a LoRa gateway. The gateway will send the data to MQTT, which is an IoT connectivity protocol. It's useful for connections in remote places where the bandwidth is scarce, as it could happen in its implementation on the university campus.

The data that has been received must be sent to a server and we have chosen Kafka, among other reasons, for its high scalability. Kafka is a storage system publisher / - Subscriber distributed, partitioned and replicated. It is a very efficient system regarding the speed of readings and writings, making it a very useful tool in our project and contributing a greater number of possible applications to the project.

Once Kafka receives the data, we can notify users who wish it through a POST request. To do this, we have developed a simple Web Service that is able to receive and respond to POST requests sent previously. This Web Service makes possible to expand the number of applications that our project can offer, since we would only have to develop the services we want to generate within the Web Service.

Something that would add more functionality to the project is that the communication is bidirectional, which means that the node is able to receive the response returned by the Web Service and, depending on the response, perform other actions. It would be possible to implement this option, since the MQTT can receive responses from the Web Service, it would only require to configure MQTT in the right way.

All these technologies have been explained and their operation has been described in the following chapters, arranged in a way that follows the process of the data; since its reception until its treatment and possible application.

In the following figure, we can observe the flow of the whole process, as mentioned above.

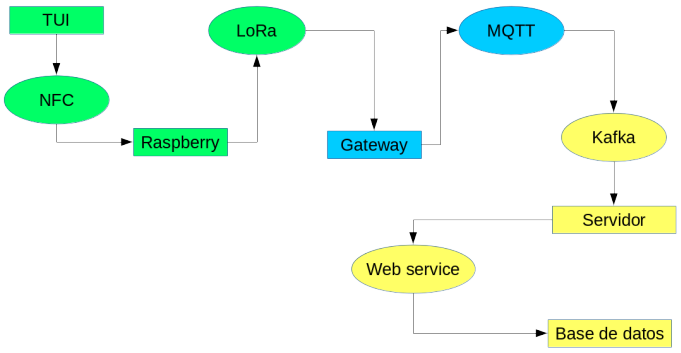

Figura B.1: Flow of the process

Apéndice C

Conclusions

Now that the project is finished, it could be said that we have achieved the goals we originally set, although, considering the development of the last few months, we cannot help but notice that these goals have frequently changed. It is not the intention of this chapter to discuss all these changes in maximum detail -since that is the purpose of other sections in this document-, but the original idea, which only comprised NFC, LoRa and Kafka, has been modified in multiple ocasions, as circumstances required to do so.
In some cases, changes were made when some unexpected limitations were found in the hardware used -as is the case of the LoRa gateway, which forced us to introduce MQTT into the data flow-, and in others when we found ourselves unable to properly configure or use a device, like, for example, the LoRa module made by Dragino, which had to be replaced for the current solution, with the added cost that implies.

As another example, some of the original ideas related to the reading of the smart cards had to be discarded, since we were not able to extract some information from the cards, due to the employed API not yet having implemented the needed functionality. This is another example of a modification that we were forced to make, but in this case, it is related to a problem that we could not, in any way foresee, and thus had to face once the project was already in development.

Eventually, we found no other option but to abandon the original plan, which would have required the node to store a read counter inside every card, so as to prevent in some way users from commiting fraud by duplicating a card.

With these and other issues that came up during the development of this project, we found ourselves forced to change, in some cases the implementation of a specific part of the network, and in others the actual final purpose of the entire project, with a direct impact on some of its use cases. However, looking at the final product, we all agree that we are satisfied with the result. It accomplishes, to a large degree, the goals that were set in the beginning, and the problems encountered have not stopped the development from continuously pushing forward until the expectations were met.

Since none of the three students had previous experience in the development of a similar project, this process has required many hours of investigation on the technologies and devices used, and thus a learning experience on many topics. And not just on the technical aspect of these protocols or the Internet of things, but also on the working methodology that a project like this requires, the implications of working with other people on the same tasks for a full year, the importance fo planning and reading carefully the necessary documentation, and especially the patience and the ability to overcome any errors and find solutions when something does not work as expected. For these reasons, we can all agree that this has been a gratifying experience and that we have learnt many important lessons.

C.1. Future work

Here is a list of some possible modifications that would improve the project in some way or another. Some of them were not integrated into the product because they exceeded the scope of this project and others were discarded during development due to some unexpected

problem we encountered:

Replace the LoRa module for a cheaper option. Although this may require the entire LoRa node to be redesigned, the cost of the current solution is too high for a large scale deployment. An alternative that was originally considered was the LoRa HAT by Dragino, but it was eventually discarded when we could not find the way to configure it to suit our needs.

Make comunication be reversible . Currently, the transmission of data only works one way, that is, from the RaspberryPi to the Webservice. Given how the infrastructure was built, it should not require too much work for a person to make the information flow both ways.

Add a read counter to the smart cards . As we previously explained, the software library we used to extract the information from the smart cards via NFC does not yet include the necessary functionality to write to the cards used in the UCM, so this feature could not be added to the final project. Adding a read counter to the cards would allow for a basic control of fraud by duplication.

Distinguish between users . Taking the UID that was read from the smart cards, the data base could keep a record of which user does it belong to. This modification, along with the previous one about making data flow both ways, is necessary to achieve the original goal of providing access control to specific areas, only allowing some users to open certain doors by using their card.

Apéndice D

Instrucciones de uso

D.1. Preparación

Para una configuración básica de la arquitectura de este proyecto, necesitaremos al menos:

- Una Raspberry Pi modelo 2 o 3.
- Un lector de tarjetas NFC EXPLORE-NFC.
- Una tarjeta con un chip NFC. A ser posible una TUI.
- Un adaptador de RaspberryPi a Arduino, o una placa equivalente con un socket XBee.
- Un módulo de transmisión LoRa Microchip RN2843.
- Una antena de 868 MHz compatible con el módulo anterior.
- Al menos 5 cables macho-hembra de conexión de pines.
- Un extensor de pines para Raspberry Pi que al menos los duplique.
- Un gateway LoRa Multitech Conduit, modelo AEP.
- Un ordenador con conexión a internet.

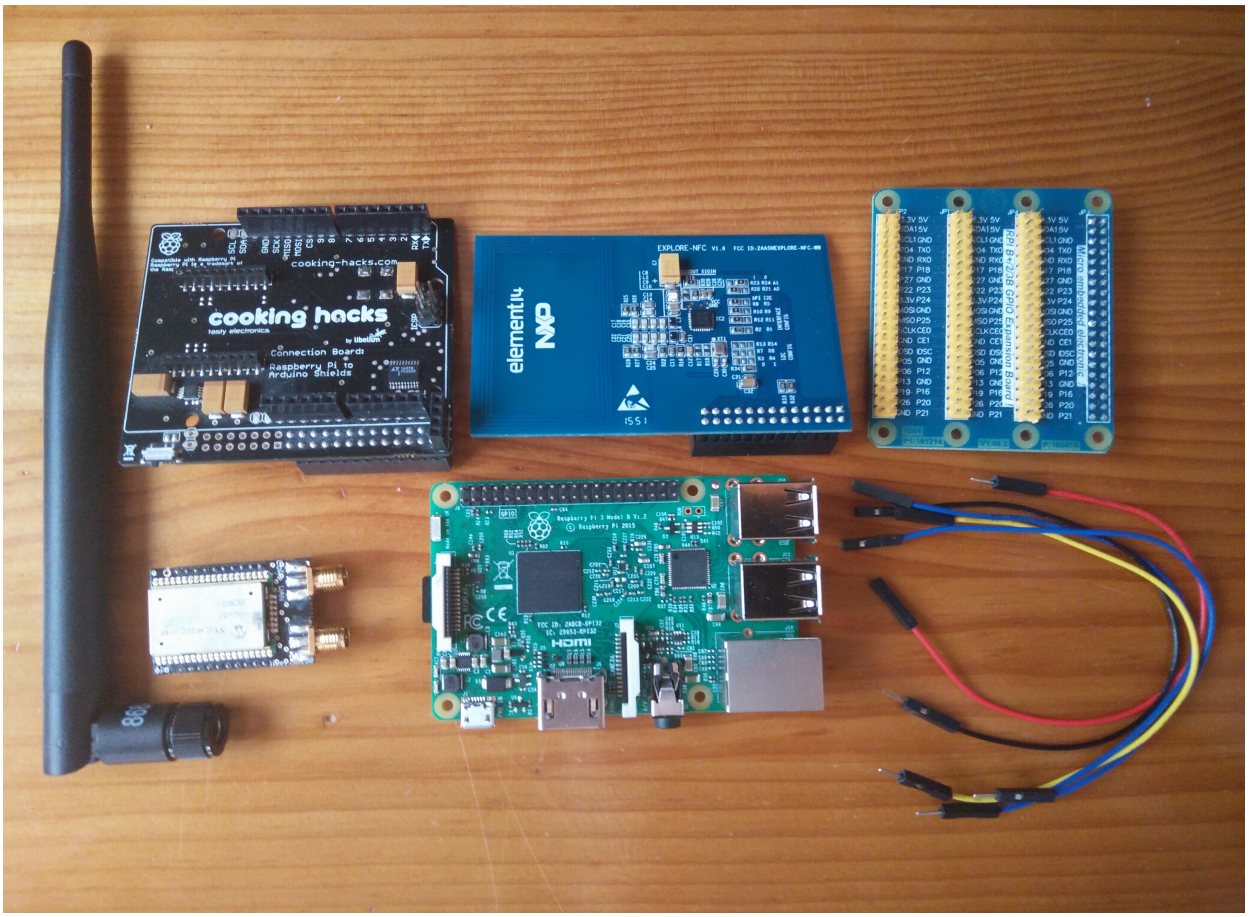

Figura D.1: Componentes necesarios para el ensamblaje

El ensamblaje de todas las piezas del dispositivo debe realizarse cuidadosamente siguiendo estos pasos. Es importante no conectar la Raspberry a la fuente de alimentación hasta que no estén todas sus extensiones conectadas:

1. En primer lugar conectamos el extensor de pines a la Raspberry, esto nos permitirá conectar el resto de las extensiones

2. A continuación se coloca el lector de NFC directamente sobre uno de los conjuntos de pines del extensor, intentando no cubrir los otros grupos.

3. Seguidamente conectamos las piezas necesarias para la comunicación LoRa. Primero enroscamos la antena a su conector correspondiente en el módulo LoRa. Si éste tiene

dos conectores, como es el caso del fabricado por Cooking Hacks, debería tener un indicador en la parte inferior de la frecuencia de antena que espera cada uno. En este caso la antena funciona en la banda de 868 MHz, por lo que la conectaremos al que tiene la etiqueta de 868/915MHz. Cuando esté conectada, colocamos el módulo LoRa en el correspondiente socket XBee de la placa adaptadora de Arduino, asegurándonos de que la antena que mirando hacia afuera.

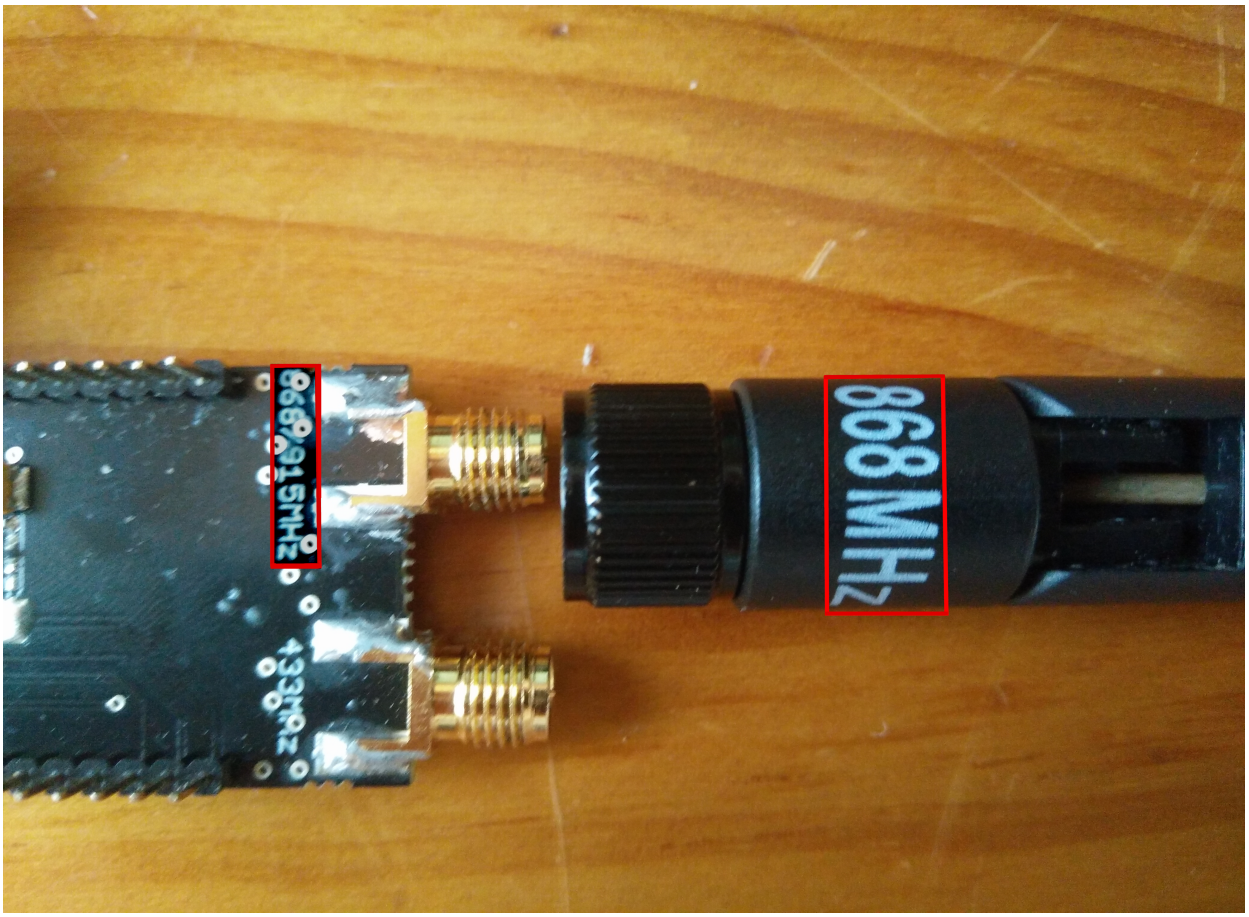

Figura D.2: Detalle de la conexión de la antena

4. Ahora que está ensambladas las piezas necesarias para LoRa, conectaremos el shield de Arduino a la Raspberry Pi. Para ello conectamos uno por uno los cables macho-hembra, enchufando un extremo en uno de los pines de un grupo libre del extensor, y el otro en su correspondiente del adaptador a Arduino. Las conexiones deben seguir

la distribución que se muestra en la siguiente tabla.

Número en Raspberry	Nombre en Raspberry	Pin de Arduino
1	3v3	3v3
6	GND	GND
8	TX0	RX0
10	RX0	TX0
17	3v3	3

Una vez que están todas las partes conectadas, el dispositivo debería tener un aspecto parecido a este:

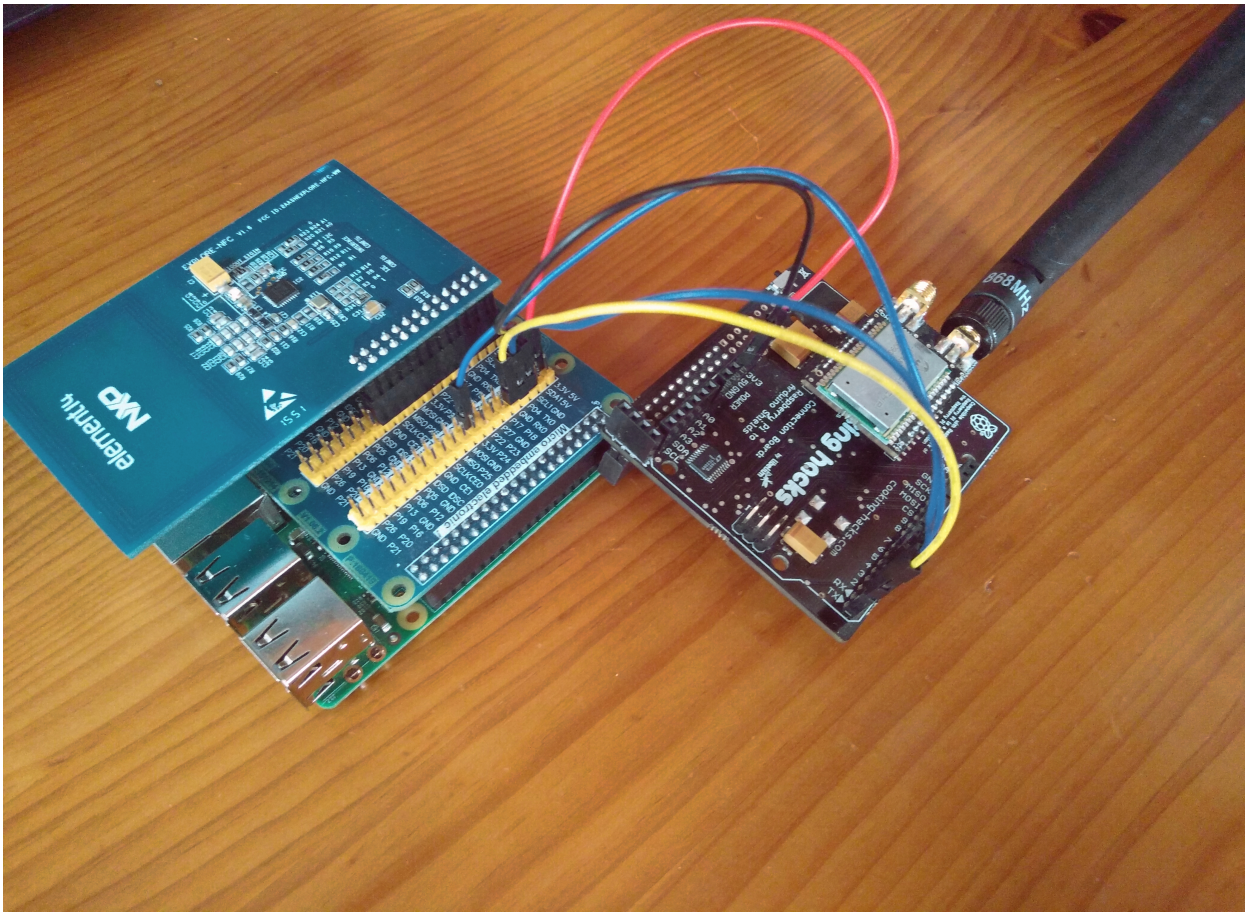

Figura D.3: Nodo ensamblado completamente

Cuando se hayan completado estos pasos se podrá conectar la Raspberry a la corriente. Omitiremos los pasos de instalación del sistema operativo, la conexión a la red y demás procesos de configuración de una Raspberry Pi, y pasaremos a asumir que el dispositivo

está operativo y accesible desde el inicio. También se asumirá durante el resto de este capítulo que el usuario tiene conocimientos básicos de un sistema GNU/Linux y que es capaz de instalar paquetes, editar texto y compilar y ejecutar programas desde una terminal.

D.2. Configuración

D.2.1. Raspberry Pi

Configuración del sistema

En primer lugar, para que la Raspberry se pueda comunicar con los otros dispositivos, será necesario habilitar las interfaces UART y SPI. Para ello editaremos /boot/config.txt, añadiendo estas líneas al final:

```
dtoverlay=pi3-miniuart-bt

enable_uart=1

dtparam=spi=on
```

Después eliminaremos de /boot/cmdline.txt cualquier referencia al puerto serie UART, de forma que si el archivo tiene un contenido similar a:

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 console=tty1 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline rootwait
```

Se transforme en (sin salto de línea):

```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4
elevator=deadline rootwait
```

Finalmente, será necesario reiniciar la Raspberry para aplicar los cambios.

Instalación de librerías

El lector de tarjetas solo requiere un paso de configuración que consiste en instalar la biblioteca necesaria para ejecutar el código como se explicará más adelante. Esta biblioteca se llama nxppy y se puede instalar de forma estándar a través de pip, usando el comando:

```
sudo pip install nxppy
```

El nodo LoRa necesita la instalación de las biblioteca arduPi y arduPiLoRaWAN, que deberemos modificar antes de poder usar, por las razones ya explicadas en el apartado**??**. Un archivo binario ejecutable ya compilado, y la versión ya modificada del código se entrega adjuntos a este documento, pero en caso de que no se tenga acceso a estos recursos, será necesario descargar la biblioteca oficial y modificarla.

NOTA: Esta modificación se ha probado con la versión oficial de la biblioteca a fecha de esta publicación: 0.4. No se asegura que pueda funcionar con versiones futuras.

NOTA2: Estos pasos están probados para la Raspberry Pi versión 2 y 3, consultar la documentación oficial en caso de utilizar una versión diferente.

En primer lugar descargamos y descomprimimos la versión oficial de la biblioteca de:
http://www.cooking-hacks.com/media/cooking/images/documentation/raspberry_arduino_shield/raspberrypi2.zip

Cuando lo hayamos hecho, modificaremos el código del archivo arduPi/arduPiUtils.h, en el que cambiaremos la definición de SOCKET_PW. Para ello buscaremos la línea que contiene:

```
#define SOCKET_PW               3
```

y la modificaremos cambiando el número de pin:

```
#define SOCKET_PW               4
```

Esta es la única modificación necesaria de la biblioteca, ahora se puede cerrar el archivo, y ejecutar el script `install_arduPi` incluido en el archivo descargado previamente.

Cuando haya terminado, podemos volver al directorio original y descargar un nuevo archivo que contiene una segunda biblioteca, llamada arduPiLoRaWAN, que utiliza la biblioteca anterior para comunicarse con el dispositivo concreto que estamos usando, el Microchip RN2843.

Descargamos arduPiLoRaWAN de la web oficial y lo descomprimimos en el mismo directorio que arduPi

`http://www.cooking-hacks.com/media/cooking/images/documentation/tutorial_kit_lorawan/arduPi_api_LoRaWAN_v1_3.zip`

Una vez que lo tenemos, podemos usar el script `cooking/examples/LoRaWAN/cook.sh` para compilar las bibliotecas junto con nuestro código. Se incluyen en el mismo directorio una serie de ejemplos de las diferentes funcionalidades de esta biblioteca, cuya función queda fuera del propósito de este proyecto. Se puede utilizar el script `cook.sh` para compilar y ejecutar el archivo adjunto a este documento llamado `lora.c`.

D.2.2. Gateway LoRa

Inicio

Asumiendo que el gateway es un Multitech Conduit modelo AEP, y que no tiene ninguna configuración previa, se pueden seguir estos pasos comenzar a usarlo. Es importante notar que, reflejando las condiciones en las que se desarrolló el proyecto, esta configuración asume que se dispone de una dirección IP pública.

1. Si el ordenador está conectado a una red WiFi, es recomendable desconectarlo.
2. Conectar el Conduit al ordenador usando un cable Ethernet.

3. Añadir, si la interfaz que corresponde al cable previamente conectado no la tiene ya, una dirección IP en la subred 192.168.2.0/24 que sea distinta a 192.168.2.1 (que es la dirección que toma el gateway por defecto).

4. Abrir un navegador y acceder al portal de configuración del gateway introduciendo http://192.168.2.1 en la barra de direcciones

   a) Es importante que la dirección incluya la parte inicial de http y no https, que no está soportado directamente por el gateway.

5. Añadir una excepción de seguridad si el navegador bloquea el acceso.

6. Introducir una contraseña y la fecha y la hora correspondientes

7. En la ventana de *Network Interface configuration*, introducir los siguientes valores (suponiendo que se use la dirección originalmente asignada al gateway de este proyecto):

   - **Type**: WAN
   - **Mode**: Static
   - **IP Address**: 147.96.21.27
   - **Mask**: 255.255.255.0
   - **Gateway**: 147.96.21.1
   - **Primary DNS Server**: 147.96.1.9
   - **Secondary DNS Server**: 147.96.2.4

8. Habilitar todas las opciones de acceso.

Finalmente la configuración debería tener el aspecto de ??. Si ha habido algún error, la configuración de acceso se puede modificar en *Administration >Access configuration*, y la configuración de direcciones accediendo a *Setup >Network Interfaces* y seleccionando el botón de editar.

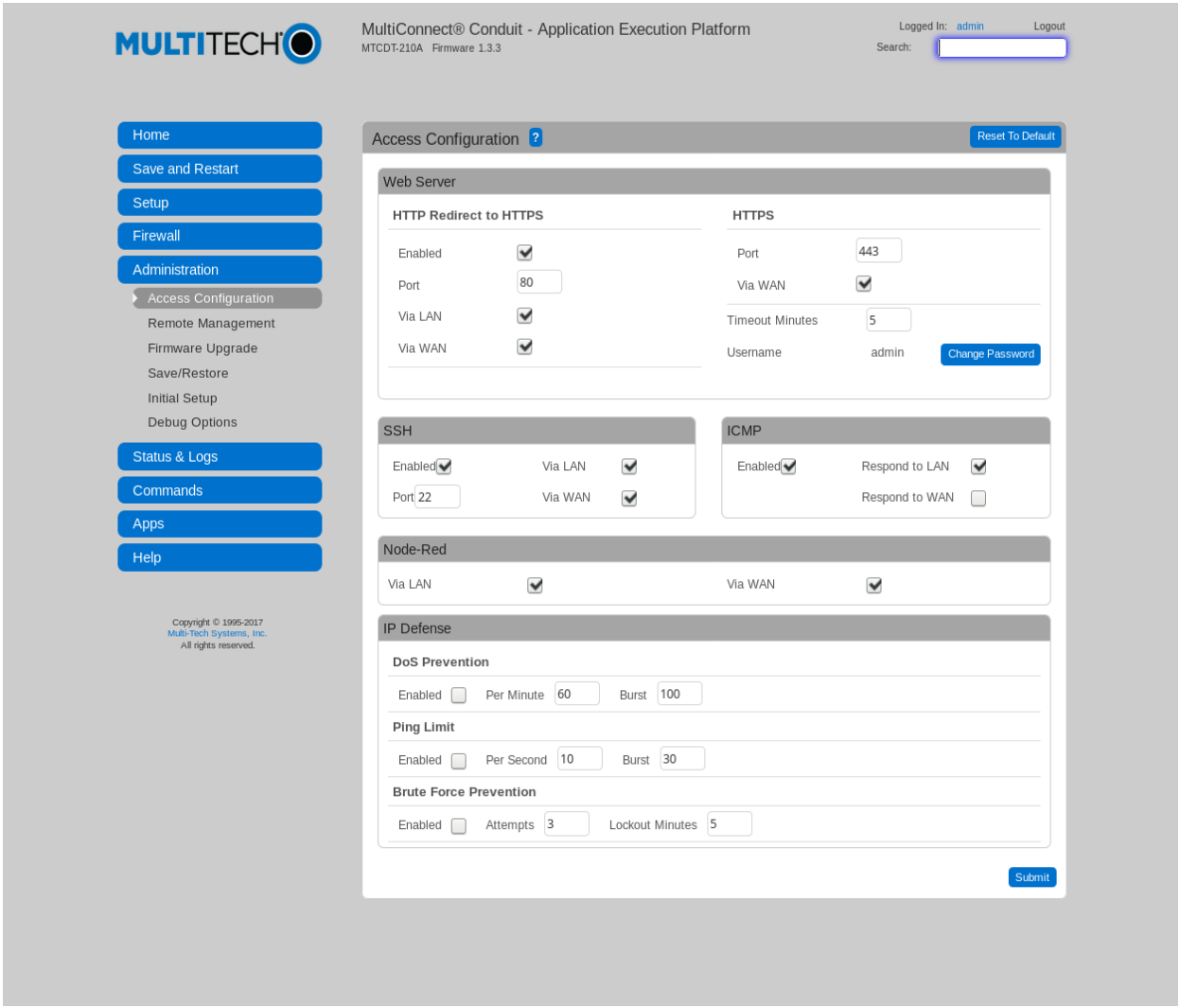

Figura D.4: Configuración de acceso

LoRa

Para configurar los parámetros de LoRa solo hace falta acceder al menú de configuración en el portal web, que debería encontrase en *Setup >LoRa Network Server*. Aquí verificamos que los siguientes parámetros tienen los valores correctos:

- El servidor LoRa está habilitado
- La banda que utiliza es la europea, de 868MHz
- El modo de operación es NETWORK SERVER y no PACKET FORWARDER

- El acceso es público

Si todos estos ajustes son correctos, pasamos a establecer las claves de LoRaWAN para que los dispositivos puedan unirse a la red con OTAA. Por asuntos de compatibilidad, es preferible cambiar *Network ID* a modo *EUI* y *Network Key* a modo *Key*. Ahora podemos introducir las claves en sus campos correspondientes. *EUI* es una cadena de 16 dígitos hexadecimales y *Network Key* una de 32. En este proyecto se usa `0001020304050607` como *EUI* y `000102030405060708091011121314l5` como *Network Key*. Es importante mantener estos valores si se pretende utilizar el código desarrollado para los nodos, o se deberá cambiar ambas claves en el archivo fuente y recompilarlo.

Cuando hayamos decidido y establecido las claves podemos seleccionar el botón *Submit* en la parte inferior para aplicar los cambios.

Si se han seguido correctamente todos los pasos, la página de configuración debería tener el aspecto de la figura ??

Node-RED

Ahora que el gateway está configurado se puede crear el flujo de Node-RED que envía los datos recibidos de LoRaWAN al servidor vía MQTT. Para ello accederemos a Node-RED usando el menú lateral y seleccionando *Apps >Launch Node-RED*. Al entrar a Node-RED se pedirá iniciar sesión con las mismas credenciales que el propio portal de configuración, esto es, con el usuario por defecto *admin* y la contraseña establecida en el paso anterior.

Al entrar a Node-RED podemos crear un nuevo flujo manualmente seleccionando y arrastrando los nodos correspondientes como se mostró en la figura **??**, o, si aún se dispone del archivo adjunto con este documento, copiar el contenido de nodered.json e importarlo seleccionando en el menú superior izquierdo *Import >Clipboard* y pegando el contenido.

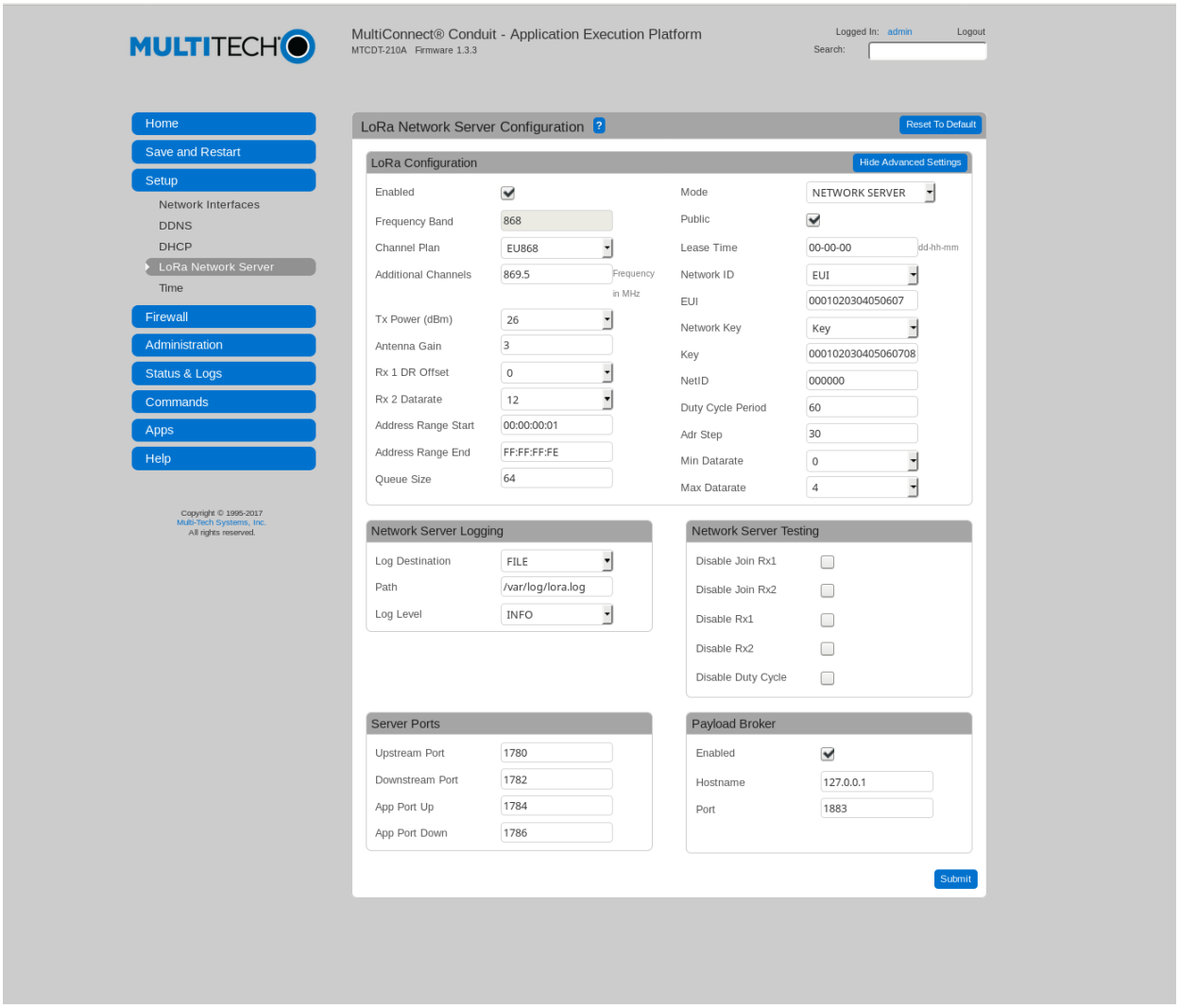

Figura D.5: Configuración de LoRa

Cuando el flujo esté creado, conviene modificar el nodo de MQTT haciendo doble click y cambiando sus parámetros. Especialmente la dirección del servidor si se está usando uno distinto al original, *taiga.dacya.ucm.es*.

Una vez que todos los parámetros sean correctos se puede hacer click en *Deploy* para guardar los cambios e iniciar la ejecución. A partir del momento en que Node-RED muestre el mensaje de confirmación, toda la información que el gateway reciba a través de LoRa se reenviará por MQTT al puerto 1883 de la máquina especificada.

D.2.3. Servidor

Dentro de la máquina virtual debemos arrancar varios procesos para el correcto funcionamiento de Kafka. En primer lugar, es necesario arrancar ZooKeeper, pues Kafka necesita que ZooKeeper esté funcionando para que pueda ser arrancado. Para arrancar ZooKeeper, iremos a la carpeta donde se encuentran todos los ficheros de instalación de Kafka (kafka_2.11-0.10.1.0), y desde ahí, ejecutamos el siguiente comando:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

A continuación, ya podremos arrancar Kafka con el siguiente comando:

```
bin/kafka-server-start.sh config/server.properties
```

Una vez Kafka está funcionando, debemos poner en funcionamiento el enlace entre MQTT y Kafka. Ejecutamos el fichero .jar correspondiente con el siguiente comando:

```
java -jar mqttKafkaBridge.jar --user tfg-nfc --pass TFGmosquitto2016
```

Una vez este enlace está activo, ya podremos recibir eventos a través de Kafka, una vez hayan pasado por MQTT. A continuación, que desplegar la función de detección de eventos y su posterior notificación mediante HTTP POST, ejecutamos el fichero KafkaTFG.jar con el comando:

```
java -jar KafkaTFG.jar
```

Con esto, todas las funcionalidades de Kafka estarían activas. Por último, nos quedaría desplegar el Web Service. Primero, debemos colocar el fichero war generado al compilar el código del Web Service en la carpeta /usr/local/apache-tomcat-9.0.0.M21/webapps Tras situar el fichero war en esa carpeta, hay que levantar el servidor web para que se despliegue el Web Service, que en este caso, es Apache Tomcat. Para ello, debemos ir a la carpeta /usr/local/apache-tomcat-9.0.0.M21/bin y ejecutar el fichero startup.sh de la siguiente forma:

```
sh startup.sh
```

Con esto, finalmente tendríamos Kafka corriendo con las funcionalidades adicionales en funcionamiento, y el Web Service desplegado a la espera de las peticiones HTTP POST.

D.3. Ejecución

Cuando todos los pasos anteriores estén completados y el sistema esté preparado para poder ejecutar la aplicación, la ejecución se reduce a arrancar `nfc.py` y `lora` desde la Raspberry Pi. Si no se conserva la versión precompilada del programa `lora`, será necesario compilar manualmente `lora.c`, usando el script `cook.sh` del que se habló detalladamente en el apartado anterior.

Ambos programas bloquearán la terminal y se ejecutarán infinitamente, por lo que deben ser lanzados independientemente en dos shell interactivas distintas. También es relevante comentar que los dos programas necesitan privilegios elevados, por lo que deberán ser ejecutados por el usuario root o a través de sudo.

Cuando se lancen, el lector de tarjetas se mantendrá activo todo el tiempo, de forma que el usuario solo deberá pasar una tarjeta o cualquier chip NFC por encima y los mensajes apropiados se enviarán por las diferentes redes hasta el servidor, sin requerir ninguna interacción más.

Apéndice E

Ejemplo de ejecución

Una vez que está todo configurado, el uso es relativamente sencillo. En este capítulo mostraremos las diferentes fases por las que pasa la información desde el nodo hasta el web-service.

En primer lugar, accedemos a la Raspberry y abrimos dos terminales. En esta demostración, se ha abierto una conexión por SSH y se ha utilizado Tmux para dividir una terminal en dos, pero no es necesario hacerlo del mismo modo. Cuando tengamos la tarjeta que queremos leer, ejecutamos el programa lector de tarjetas como root, por ejemplo usando la orden: `sudo python nfc.py`. Cuando éste muestre por la pantalla un mensaje diciendo que está listo para leer, pasamos la tarjeta por el lector y ejecutamos el otro programa (también como root) en la otra terminal. Éste se encargará de enviar los datos leídos, e imprimirá información sobre su funcionamiento mientras lo haga. Si el lector ha dejado de leer tarjetas, el programa de LoRa terminará y apagará el módulo debidamente. Un ejemplo de esta ejecución se muestra en la figura E.1

Ahora podemos ir al gateway y comprobar que los datos están llegando efectivamente. Para ello entramos en Node-RED como se explicó en el capítulo anterior, introducimos nuestras credenciales y añadimos, si no existe ya, un nodo de tipo debug al final del flujo, como se muestra en la figura E.2. Si ha hecho falta añadirlo, hacemos click en *Deploy* y esperamos

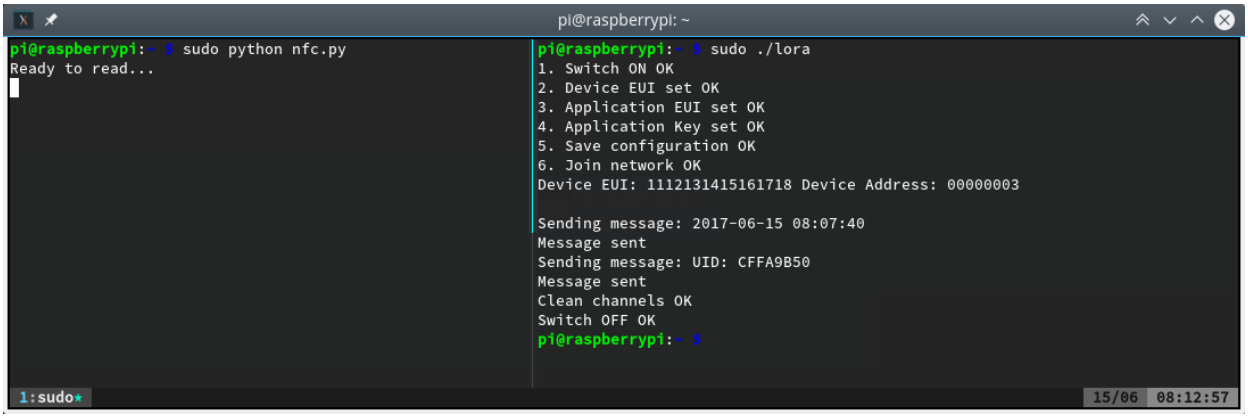

Figura E.1: Ejecución en la Raspberry

al mensaje de confirmación. Una vez que esté añadido este nodo, los mensajes recibidos aparecerán en el menú lateral, en la pestaña *debug*, e incluirán la información exacta que se enviará al servidor por MQTT.

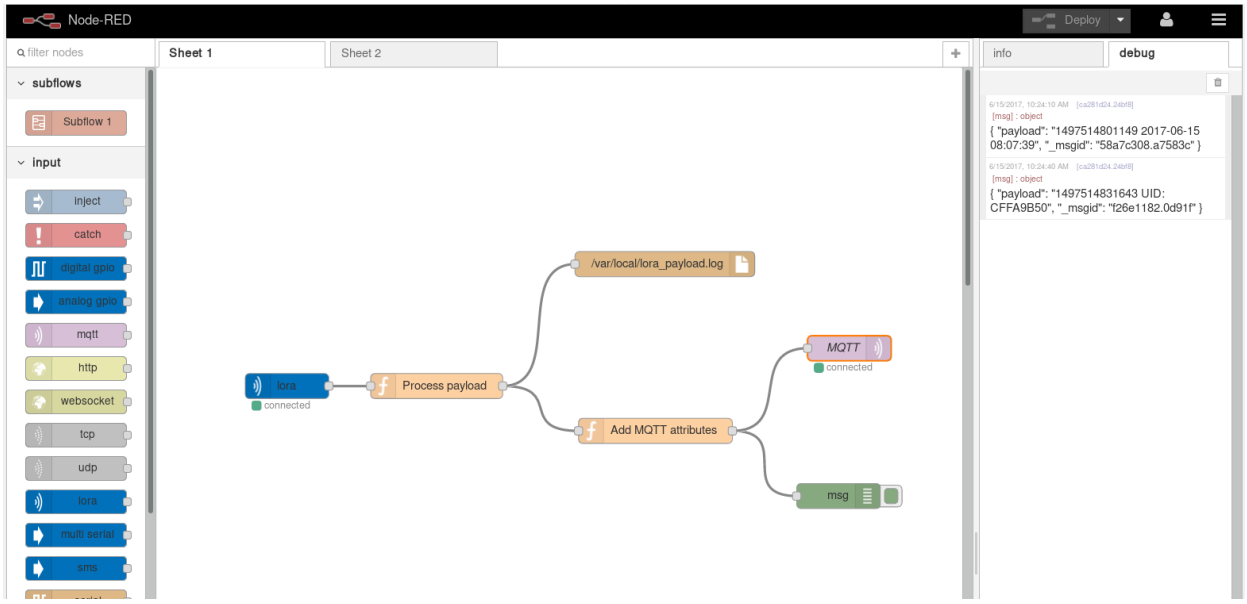

Figura E.2: Mensajes en Node-RED

Los mensajes enviados por MQTT al servidor se recibirán en el servidor Kafka. Para arrancarlo, como se ha explicado antes, es necesario tener Kafka instalado en la máquina y arrancar, por orden: Zookeeper, Kafka, el bridge de MQTT a Kafka y la aplicación que

hemos desarrollado. Para ello abriremos 4 terminales y ejecutaremos en cada una uno de estos comandos. Es importante seguir el orden y esperar a que cada script haya terminado su proceso de inicio antes de lanzar el siguiente.

1. `kafka/bin/zookeeper-server-start.sh config/zookeeper.properties`
2. `kafka/bin/kafka-server-start.sh config/server.properties`
3. `java -jar 'Conexion Kafka/mqttKafkaBridge.jar' --user tfg-nfc --pass TFGmosquitto20`
4. `java -jar 'Conexion Kafka/KafkaTFG.jar'`

El orden de ejecución se muestra en la figura E.3.

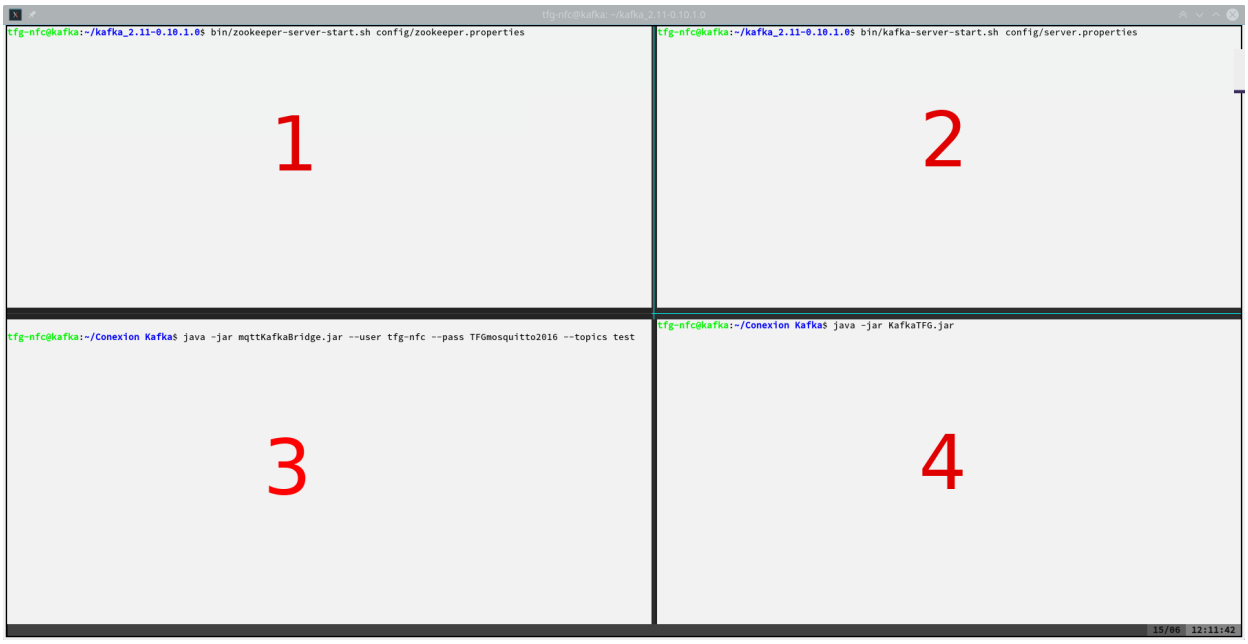

Figura E.3: Preparación de la ejecución del servidor Kafka

Si todo se ejecuta correctamente, recibiremos los mensajes en la terminal número 4, que también dará información sobre las peticiones HTTP que ha hecho al web service. Un

ejemplo de esta ejecución se muestra en la figura E.4.

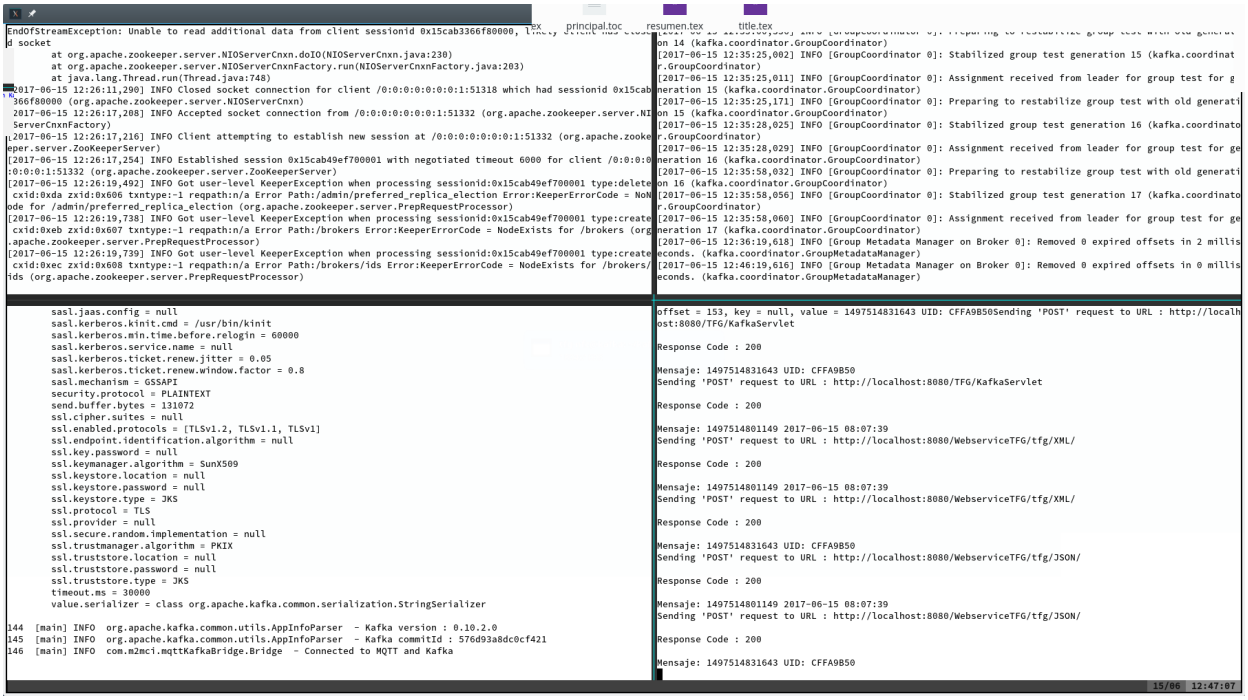

Figura E.4: Recepción de los mensajes MQTT y envío al web service